OGC® DOCUMENT: 22-014

External identifier of this OGC® document: http://www.opengis.net/doc/PER/T18-D008

TESTBED-18: KEY MANAGEMENT SERVICE ENGINEERING REPORT

ENGINEERING REPORT

PUBLISHED

Submission Date: 2022-11-05
Approval Date: 2022-12-15
Publication Date: 2023-01-05
Editor: Andreas Matheus

CONTENTS

LIST OF TABLES

LIST OF FIGURES

I EXECUTIVE SUMMARY

Data Centric Security (DCS) defines the principles for applying security to the data itself. The confidentiality of the data is ensured by applying encryption based on cryptographic keys. Similarly, integrity and authenticity can be ensured based on cryptographic keys.

This document defines a cryptographic key data model, a flexible API, and conformance classes to implement a Key Management Service.

The API of the KMS is separated into logical interfaces to generate, register, update, and delete keys for encrypting / decrypting geospatial data (Data Encryption Key or DEK) and keys for protecting the DEK (Key Encryption Key or KEK) as well as Public Key (PK). The DEK interface focuses on the CRUD of symmetric keys that are used for the actual data ciphering. The KEK interface focuses on the CRUD of asynchronous keys that are used to protect DEKs (key wrapping) in formats, where the data and the actual data encryption key are stored together. The PK interface focus on the exchange of public keys to support signature validation.

During the OGC Testbed 18, a reference implementation of the described KMS was used to exchange confidential and integrity protected catalog metadata via the OGC API Records.

Concluding from the results of OGC Testbed 18 in the Secure and Asynchronous Catalog task, one can say that the described KMS has proven to be a paramount OGC building block for enabling Data Centric Security.

II KEYWORDS

The following are keywords to be used by search engines and document catalogues.

key management service, KMS, API, DCS, security, testbed, web service, encryption, digital signature

III SECURITY CONSIDERATIONS

Regardless of the encryption solution, the weakest point is always the safety of the DEK; either ensured via the KEK in the first case or by the KMS in the second case. This is because for the self-contained solution where the DEK is available with the encrypted data, this solution can only be considered secure as long as it takes to brute force the KEK. A brute-forced KEK gives access to the DEK and thereby enables the decryption of the data. Therefore, this solution requires the use of an extremely secure (large key length) KEK to further protect access to the encrypted product. Because the cracking time of a key highly depends on computing speed and power, one should only use this solution in real disconnected environments where physical access to the device(s) — in addition to network-based access — can be controlled. Different literature is available that publish the expected mean time to cracking a particular key based on a given strength and algorithm. Any existence of the self-contained product longer than this average cracking time should be interpreted as "data exists in the clear now."

Additional security considerations specific for the Key Management Service are outlined in Clause 8.

IV SUBMITTERS

All questions regarding this submission should be directed to the editor or the submitters:

NAME	AFFILIATION	ROLE
Andreas Matheus	Secure Dimensions GmbH	Editor

V ABSTRACT

This OGC Testbed 18 Engineering Report describes the Data Model and API of a Key Management Service (KMS) that supports the flexible but secure exchange of cryptographic keys for applying confidentiality and integrity protection to geographic information. The described KMS is based on the design and implementation from previous OGC Testbeds 16[1] and 17[2].

[1]Aleksandar Balaban

[2] Aleksandar Balaban, Andreas Matheus

1

SCOPE

1 SCOPE

This OGC Testbed 18 Engineering Report defines a Key Management Service that supports the secure management and exchange of cryptographic key material. This document defines a data model and an API to register, create, update, and delete symmetric and asymmetric keys for the purpose of applying confidentiality and integrity to geospatial information.

2

NORMATIVE REFERENCES

2 NORMATIVE REFERENCES

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

Open API Initiative: OpenAPI Specification 3.0.2, 2018 https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.2.md

R. Fielding, J. Reschke (eds.): IETF RFC 7231, *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC Publisher (2014). https://www.rfc-editor.org/info/rfc7231.

R. Fielding, J. Reschke (eds.): IETF RFC 7235, *Hypertext Transfer Protocol (HTTP/1.1): Authentication*. RFC Publisher (2014). https://www.rfc-editor.org/info/rfc7235.

M. Jones: IETF RFC 7517, *JSON Web Key (JWK)*. RFC Publisher (2015). https://www.rfc-editor.org/info/rfc7517.

M. Jones: IETF RFC 7518, *JSON Web Algorithms (JWA)*. RFC Publisher (2015). https://www.rfc-editor.org/info/rfc7518.

M. Jones, J. Bradley, N. Sakimura: IETF RFC 7519, *JSON Web Token (JWT)*. RFC Publisher (2015). https://www.rfc-editor.org/info/rfc7519.

M. Jones, A. Nadalin, J. Bradley, C. Mortimore: IETF RFC 8693, *OAuth 2.0 Token Exchange*. RFC Publisher (2020). https://www.rfc-editor.org/info/rfc8693.

M. McGloin, P. Hunt: IETF RFC 6819, *OAuth 2.0 Threat Model and Security Considerations*. RFC Publisher (2013). https://www.rfc-editor.org/info/rfc6819.

W3C cors, *Cross-Origin Resource Sharing*. https://www.w3.org/TR/cors/.

W3C did-cbor-representation, *The Plain CBOR Representation v1.0*. https://www.w3.org/TR/did-cbor-representation/.

eXtensible Access Control Markup Language (XACML) Version 3.0, OASIS, 2013, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html

Geospatial eXtensible Access Control Markup Language (GeoXACML), OGC, 2011, https://portal.ogc.org/files/?artifact_id=42734

3

TERMS, DEFINITIONS AND ABBREVIATED TERMS

3 TERMS, DEFINITIONS AND ABBREVIATED TERMS

This document uses the terms defined in OGC Policy Directive 49, which is based on the ISO/IEC Directives, Part 2, Rules for the structure and drafting of International Standards. In particular, the word “shall” (not “must”) is the verb form used to indicate a requirement to be strictly followed to conform to this document and OGC documents do not use the equivalent phrases in the ISO/IEC Directives, Part 2.

This document also uses terms defined in the OGC Standard for Modular specifications (OGC 08-131r3), also known as the ‘ModSpec’. The definitions of terms such as standard, specification, requirement, and conformance test are provided in the ModSpec.

For the purposes of this document, the following additional terms and definitions apply.

3.1. Terms and definitions

3.1.1. **CRUD (in the context of KMS)**

The HTTP principle *C*reate, *R*ead, *U*pdate and *D*elete is mapped to different HTTP methods as defined in IETF RFC 7231. In REST interface design, the terms Create, Read, Update, and Delete determine the operation on a resource that is triggered by an HTTP method. It should not be confused with the operations on keys. For example, the registration of a key is de facto triggered by the HTTP method `POST` or `PUT`. The term *registration* is preferred in this document to not confuse the generation of a key, aka *creation* of a key.

3.1.2. **DCS Application**

An application that consumes DCS assets such as encrypted and/or digitally signed data.

3.1.3. **DCS Consumer**

The party that triggers the production of, or downloads, previously produced DCS assets.

3.1.4. **DCS Service**

A web-accessible implementation that supports the production of DCS assets.

3.1.5. **DCS Task**

A back-office functionality for producing DCS assets.

3.2. **Cryptographic Key**

A cryptographic key (or key for short) is a random secret used in combination with an algorithm (cipher) to transform readable information into unreadable information and vice versa.

3.3. Abbreviated terms

DCS	Data Centric Security
DEK	Data Encryption Key
DPoP	Demonstrating Proof of Possession
JWE	JSON Web Encryption
JWK	JSON Web Key
JWS	JSON Web Signature
JWT	JSON Web Token
KEK	Key Encryption Key

KMS	Key Management Service
kurl	A custom JWE header invented in OGC Testbed initiative 16 to represent a key resource via a resolvable URI
KVP	Key Value Pair
PK	Public Key

4

INTRODUCTION

4 INTRODUCTION

In the context of Data Centric Security (DCS), the use of encryption is important to achieve confidentiality. Encryption itself is based on the use of (symmetric) Data Encryption Keys (DEKs) that jumble of the data into an unreadable format. The use of (asymmetric) Key Encryption Keys (KEKs) is typically used to enable protection of the DEK. The use of digital signatures is important to ensure integrity and authenticity of data assets. Integrity is based on a private / public (asymmetric) key pair. Authenticity can be achieved when using X.509 certificates that bundle the public key with identity information.

The following encryption use cases are important in the context of this Engineering Report.

- For the first case, the protected DEK is stored with the encrypted data.
- For the second case, the DEK is negotiated out of band between the involved parties — so the DEK is referenced from the encrypted data.

Applying the first case creates a DCS product that is self-contained: encrypted data and the wrapped DEK are stored together. To protect the actual DEK, a KEK is used that can only be decrypted with the associated private key and optionally can only be activated by providing a PIN or some other credential in addition (the private key is password protected). Therefore, a self-contained DCS product is best used in disconnected, or offline environments. The limitation is obvious: only the entity with the correct private key can decrypt the DEK. Sharing of private keys is not allowed and therefore the slate of recipients must be known beforehand when creating a self-contained DCS product.

The second DCS product, where the encrypted data and the DEK are separated, has the advantage over the first solution in that it allows an independent sharing of the actual encrypted data and the DEK. This flexibility comes with the requirement of network connectivity to a key management service or the availability of another secure device like a protected USB stick.

A network accessible service, Key Management Service (KMS), is a flexible solution to manage the DEK and the KEK independent from the encrypted data. Controls can be enabled at the KMS that regulate access to the keys under given (but perhaps adaptable) conditions. The use of network based key management provides flexibility but also requires that a client application has network connectivity to obtain the required DEKs.

Applying digital signatures as a means for integrity or authenticity requires the ability to manage the public keys (PKs). Well established methods for rolling out X.509 certificates exist in Public Key Infrastructures, but the use of public keys and X.509 certificates within DCS requires an interoperable, flexible, and trusted access.

The use of a KMS has the advantage in that the exchange of a DEK (and KEK) as well as PK to applications and users can be controlled in a flexible way. An access control concept must exist that controls the CRUD to keys to ensure key safety. Adopted access control aspects might be relevant because the use of DEK, KEK, and PK are so essentially different.

An API for a KMS that allows the management of DEKs and KEKs for the encryption of geographic information and the location of the user has one additional requirement over any other mainstream IT solution: the location of the encrypted resource, the location of the device (or application), and the user are potential characteristics that determine the access conditions. In that sense, the KMS business logic must provide the ability to enforce spatio-temporal access conditions on the DEK and perhaps on the KEK.

This Engineering Report introduces a data model, the API of a Key Management System for DEK, KEK, and PK as well as conformance classes to ensure interoperability. Aspects of a typical business logic are also discussed.

4.1. KMS Interfaces and DCS Architecture

A Key Management API cannot be designed in isolation. The DCS architecture as illustrated in Figure 1 can be split into essentially two different methods to apply DCS.

- For the first method, the DCS Consumer is the active party that triggers the data assets to be produced. Any DEK, generated by the DCS Service and registered with the KMS, is owned by the DCS Consumer. Therefore, the DCS Consumer can determine with whom to share the keys using the Key Management Endpoints of the KMS.
- For the second method, a DCS Producer leverages some DCS tasks in the own network to create encrypted and/or digitally signed data assets. The involved DEKs and PKs are owned by the DCS Producer and are registered with the KMS. After the DCS Consumer has retrieved the DCS data assets, the DCS Application interacts with the KMS to obtain the required key(s) for decryption and validation of digital signature(s). The DCS Producer controls access to the keys via the Key Management Endpoints of the KMS.

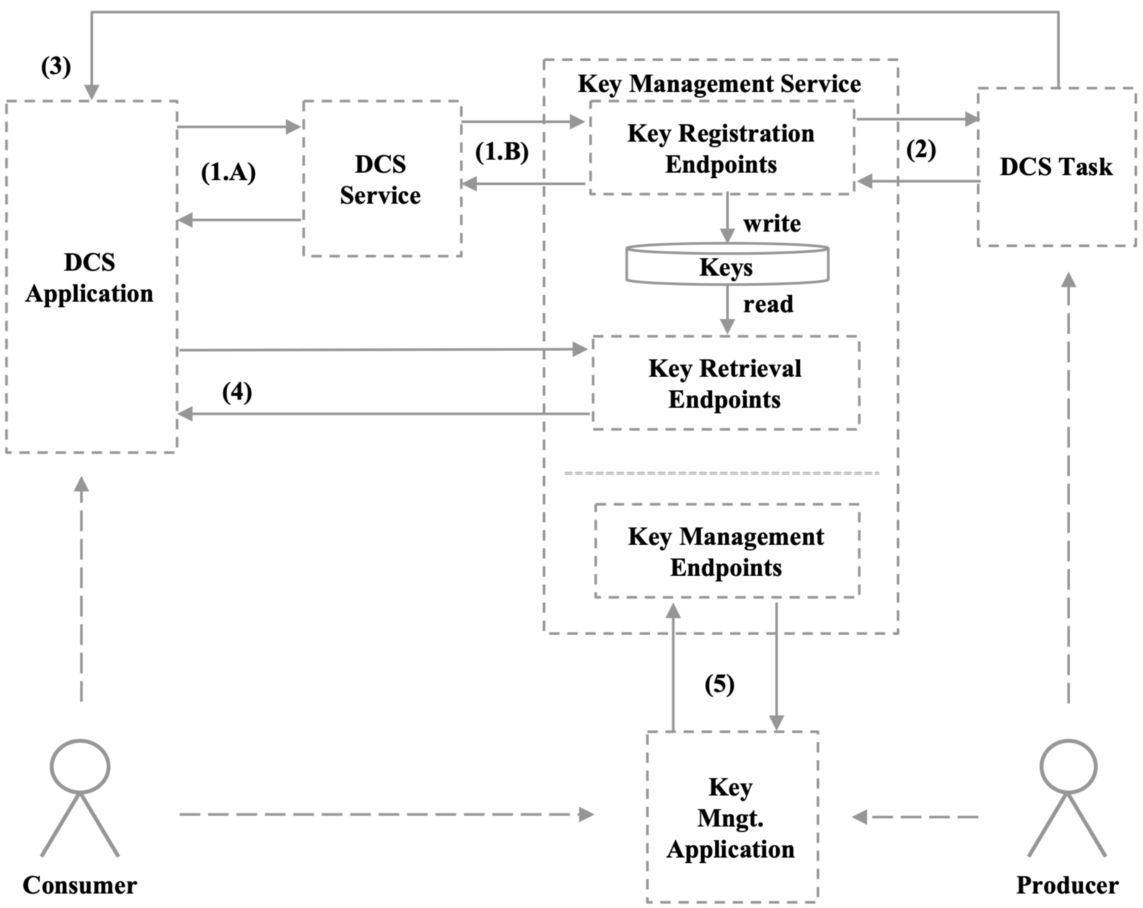


Figure 1 — DCS OGC Testbed 18 DCS Architecture

4.1.1. DCS Consumer Initiates Production of DCS Assets

The protocol labelled `(1.A)` supports different interactions with the DCS Service. Some interactions via `(1.A)` require interactions via `(1.B)`. These interactions imply different requirements to the KMS interface labelled `(1.B)`.

Interactions via `(1.A)` that **do not** require interaction via `(1.B)` include the following.

- If the request `(1.A)` includes the DEK in the clear (not encrypted by a KEK), the DCS Service will use the DEK from the request. Assuming the DEK from the request is "good enough" for encryption, there is no need for the DCS Service to interact with the KMS.
- If the request `(1.A)` includes a protected DEK (encrypted by a KEK), the DCS Service will use its private key to decrypt the DEK. This implies that the DEK was protected via the public key of the DCS Service.

Interactions via (1.A) that **do** require interactions via `(1.B)` include the following.

- If the request `(1.A)` includes a protected DEK (encrypted by a KEK), and the KEK was previously registered with the KMS (so the DCS Service's public key was not used), the DCS Service will interact with the KMS to obtain the KEK (private part) to be able to decrypt the DEK. This requires that the KEK's audience includes the DCS Service; otherwise, the KMS would refuse the DCS Service's request to obtain the KEK.

- If the request `(1.A)` includes a DEK by reference (using the JWE headers `kid` or `kurl`) to a previously created and registered DEK, the DCS Service will fetch the DEK from the KMS. This requires that (i) the DEK is accessible to the DCS Service if the DCS Service uses its own access token to interact with the KMS; or (ii) the DCS Service can re-use the access token from `(1.A)` to interact with the KMS via `(1.B)`. This access token re-use is only possible if its audience includes the KMS. For example, in the case that the access token is bound to the DCS Client and DCS Service only, the KMS must reject the token as it is not a white-listed receiver for that key. It is dangerous to accept access tokens that doe not match the own audience as these tokens may include scopes and additional claims that erroneously may grant access.
- If the request `(1.A)` does not include a DEK, but a public part from a **KEK by value**, the DCS Service will create one or multiple DEKs to encrypt the requested data. The DCS Service encrypts the DEK with the public key from the request and includes it into the response along with the encrypted data.
- If the request `(1.A)` does not include a DEK but a public part from a **KEK by reference**, the DCS Service will create one or multiple DEKs to encrypt the requested data. But in order to encrypt the DEK, the DCS Service must fetch the public part from the KEK from the KMS. As access to public keys is not restricted by the KMS, this can easily be achieved. The DCS Service encrypts the DEK with that public key fetched from the KMS and includes it into the response along with the encrypted data.
- If the request `(1.A)` does not include a DEK and also does not include a KEK (public part), neither by value nor by reference, the DCS Service will create one or multiple DEKs and will use the protocol with the KMS initiated via `(1.B)` to register these keys with the KMS. At that point, it must be ensured that the DCS Service is allowed to register the DEK on behalf of the DCS Consumer with the KMS and that the sole ownership over the registered keys remains with the DCS Consumer.

4.1.2. DCS Producer Initiates production of DCS Assets

Interactions via (2) can be used by the DCS Producer to register DEKs and PKs that were generated by the DCS Task to create DCS data assets.

Interactions via (3) can be used by the DCS Consumer to fetch the DCS data assets. It is also possible that the DCS Producer pushes the DCS data assets to the DCS Consumer. The DCS Consumer Application can fetch required keys via (4).

4.1.3. DCS Consumer Obtains Keys

Interactions via `(4)` allow the DCS Application to fetch a DEK to decrypt data and to fetch a PK for signature verification.

4.1.4. DCS Consumer Manages Their Keys

Interactions via `(5)` symbolize the possibilities to create / update / delete keys and to update access conditions for keys. The Key Management Applications can be used by the DCS Consumer and the DCS Producer to manage their keys. The Key Management Application must be trusted at a very high level as it can execute UPDATE, PATCH, and DELETE on a DEK or a KEK which is not possible via the `(1.B)` or `(2)` API. These APIs are only allowed to register or generate a key.

5

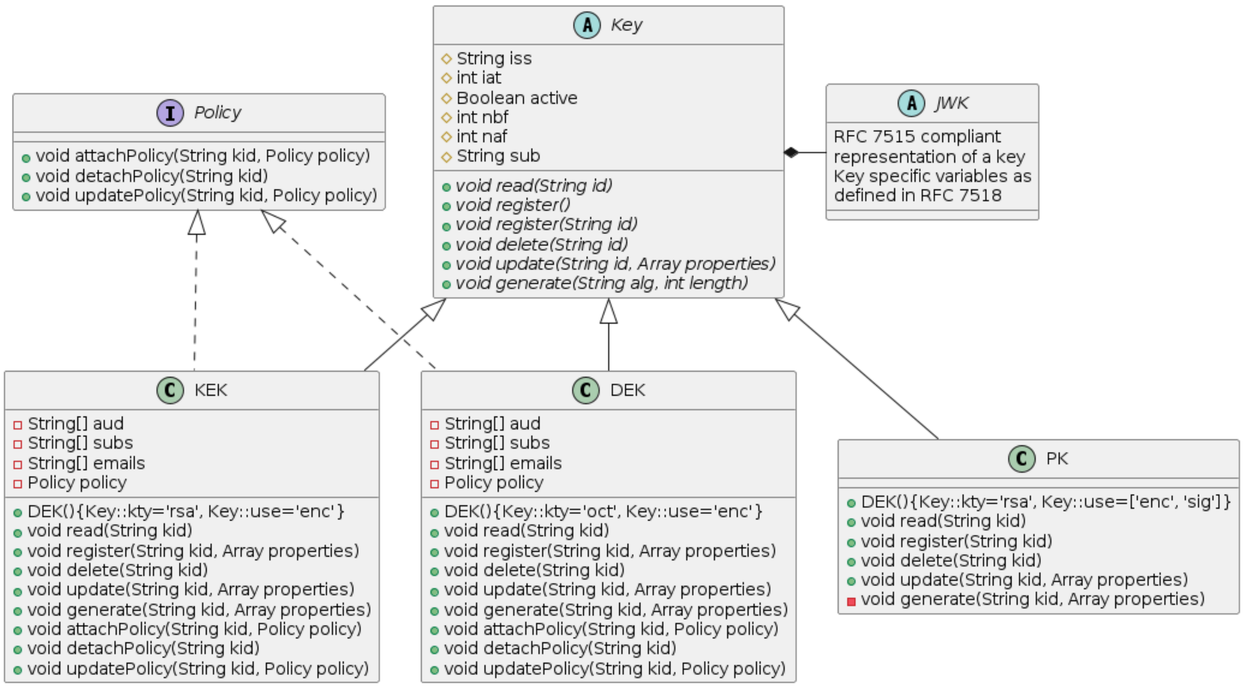
DATA MODEL, OPERATIONS AND KEY REPRESENTATION

5 DATA MODEL, OPERATIONS AND KEY REPRESENTATION

The KMS can operate on different types of keys. The data model is relevant to understanding the internal and external representation. The internal representation aligns with the business logic and the external representation aligns with the HTTP request and response structure.

5.1. Data Model

The key data model is quite simple. The core uses the properties from the JWK ([rfc7515]) representation. The abstract class `Key` defines the basic properties of a key. Each supported key type (`DEK`, `KEK`, and `PK`) is represented by a specialization. The implementation of the `Policy` interface enables the compliance to the `Policy` Conformance Class.



A Key
String iss
int iat
Boolean active
int nbf
int naf
String sub
void read(String id)
void register()
void register(String id)
void delete(String id)
void update(String id, Array properties)
void generate(String alg, int length)
I Policy
void attachPolicy(String kid, Policy policy)
void detachPolicy(String kid)
void updatePolicy(String kid, Policy policy)
A JWK
RFC 7515 compliant representation of a key Key specific variables as defined in RFC 7518
C KEK
String[] aud
String[] subs
String[] emails
Policy policy
DEK(){Key::kty='rsa', Key::use='enc'}
void read(String kid)
void register(String kid, Array properties)
void delete(String kid)
void update(String kid, Array properties)
void generate(String kid, Array properties)
void attachPolicy(String kid, Policy policy)
void detachPolicy(String kid)
void updatePolicy(String kid, Policy policy)
C DEK
String[] aud
String[] subs
String[] emails
Policy policy
DEK(){Key::kty='oct', Key::use='enc'}
void read(String kid)
void register(String kid, Array properties)
void delete(String kid)
void update(String kid, Array properties)
void generate(String kid, Array properties)
void attachPolicy(String kid, Policy policy)
void detachPolicy(String kid)
void updatePolicy(String kid, Policy policy)
C PK
DEK(){Key::kty='rsa', Key::use=['enc', 'sig']}
void read(String kid)
void register(String kid)
void delete(String kid)
void update(String kid, Array properties)
void generate(String kid, Array properties)

Figure 2 — Data Model

NOTEThe class `PK` implements the operation `generate()` with private visibility. This ensures that the class implementation is complete and there is no `generate()` operation for the public key API.

5.2. Key Variables and Their Meaning

The key variables are based on extracting meaning from other specifications where possible.

Table 1 — Meaning of common variables

VARIABLE	TYPE	DESCRIPTION
`kid`	String	An identifier for the key unique at the KMS as defined in RFC 7517, section 4.5
`active`	Boolean	If `true` the key can be accessed via the API. Default `false`
`naf`	Integer	The seconds since the epoch until this `kid` can be accessed via the API. Meaning according to RFC 7519, section 4.1.5 by replacing "before" with "after"
`nbf`	Integer	The seconds since the epoch after which this `kid` can be accessed via the API. Meaning according to RFC 7519, section 4.1.5
`iat`	Integer	The seconds since the epoch when this `kid` was registered. Meaning according to RFC 7519, section 4.1.6
`aud`	Array of Strings or URIs	All identifiers of application identifiers (e.g. OAuth2 `client_id`) that can access and use the key. Meaning according to RFC 7519, section 4.1.3
`subs`	Array of Strings or URIs	All identifiers of users (e.g OAuth2 `user_id`) that can access and use the key
`iss`	String or URI	The unique identifier (e.g. OAuth2 `client_id`) of the application that originally created the key. Meaning according to RFC 7519, section 4.1.1

Table 2 — Meaning of DEK variables

VARIABLE	TYPE	DESCRIPTION
`k`	String	Key secret encoded according to RFC 7518, section 6.4.1
`kty`	String	Fixed value `oct` to represent a symmetric key
`alg`	String	Value one of {`A128CBC-HS256`, `A192CBC-HS384`, `A256CBC-HS512`, `A128GCM`, `A192GCM`, `A256GCM`} as defined in RFC 7518, section 4.1 for symmetric keys
`use`	String	`enc`

Table 3 — Meaning of KEK variables

VARIABLE	TYPE	DESCRIPTION
`kty`	String	Fixed value `rsa` to represent an asymmetric key
`alg`	String	As defined in RFC 7518, section 4 for DEK wrapping (encryption)
`use`	String	`enc`
`key_ops`	String	["wrapKey", "unwrapKey"]

NOTE All variables to define a private and public key are allowed as defined in RFC 7518.

Table 4 — Meaning of PK variables

VARIABLE	TYPE	DESCRIPTION
`n`	String	The modulus value for the RSA public key as defined in RFC 7518
`kty`	String	Fixed value `rsa` to represent an asymmetric key
`alg`	String	As defined in RFC 7518, section 4 for asymmetric keys
`use`	String	`enc` | `sig`
`key_ops`	String	`wrapKey` if use == enc | `verify` if use == sig

NOTE No variables are allowed to define the private key with which the public key is associated.

5.3. Operations

5.3.1. Key Read

HTTP method `GET` fetches a key.

5.3.2. Key Bulk Read

HTTP method `GET` fetches a key-set.

5.3.3. Key Registration

The registration of a key with the KMS is the operation that triggers the key's representation to be stored at the KMS[3]. The KMS API protocol supports two HTTP methods for key registration as follows.

- HTTP method `POST` can register one or multiple keys where the KMS business logic generates the key identifier `kid`.
- HTTP method `PUT` can register one single key where the key identifier `kid` is provided with the HTTP request.

5.3.4. Key Bulk Registration

The registration of a key-set.

5.3.5. Key Generation

The generation of a key, aka *creation* of a key, is the operation that triggers the KMS implementation to create a key secret and eventually other key properties, based on the information received with the HTTP method `POST` or `PUT`.

For example, the key generation operation is triggered if the HTTP request to register a DEK does not contain the key's secret. For using the IETF RFC 7517 compliant representation of a key, this means that the property `k` is missing.

5.3.6. Key Activation / Deactivation

The operation that makes a key accessible via the API. The actual HTTP CRUD operations are controlled via the KMS business logic.

In similar semantics to the activation, the deactivation disables API access to the key.

5.3.7. Key Bulk Generation

The generation of a key-set.

[3]The means of storage are implementation specific.

5.3.8. Key Update

The operation that makes changes to the key's representation as JWT or JWE is limited to certain properties that control access to and usage of the key. The properties `kid`, `k`, `kty`, `alg`, and `use` **cannot** be changed.

5.3.9. Key Deletion

The operation that removes the key from the KMS storage[4]

5.4. Key Representation

The KMS API only supports one mandatory key encoding: JWK as defined in IETF RFC 7517. Even though the key representation is using JWK, the actual representation of the key varies depending on the requested content-type: JWT or JWE.

5.4.1. JWK representation of a DEK and KEK

```
{
  "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
  "alg": "A128GCM",
  "kty": "oct",
  "k": "J_W99Qhw5gbP72YpmA60Kg",
  "use": "enc"
}
```

Figure 3 — JWK example of a DEK (symmetric key):

```
{
      "kty":"RSA",
      "kid":"juliet@capulet.lit",
      "use":"enc",
      "n":"t6Q8PWSi1dkJj9hTP8hNYFlvadM7DflW9mWepOJhJ66w7nyoK1gPNqFMSQRy
           O125Gp-TEkodhWr0iujjHVx7BcV0llS4w5ACGgPrcAd6ZcSR0-Iqom-QFcNP
           8Sjg086MwoqQU_LYywlAGZ21WSdS_PERyGFiNnj3QQlO8Yns5jCtLCRwLHL0
           Pb1fEv45AuRIuUfVcPySBWYnDyGxvjYGDSM-AqWS9zIQ2ZilgT-GqUmipg0X
           OC0Cc20rgLe2ymLHjpHciCKVAbY5-L32-lSeZO-Os6U15_aXrk9Gw8cPUaX1
           _I8sLGuSiVdt3C_Fn2PZ3Z8i744FPFGGcG1qs2Wz-Q",
      "e":"AQAB",
      "d":"GRtbIQmhOZtyszfgKdg4u_N-R_mZGU_9k7JQ_jn1DnfTuMdSNprTeaSTyWfS
           NkuaAwnOEbIQVy1IQbWVV25NY3ybc_IhUJtfri7bAXYEReWaCl3hdlPKXy9U
```

[4] A real removal is not recommended by implementation specific.

```
      vqPYGR0kIXTQRqns-dVJ7jahlI7LyckrpTmrM8dWBo4_PMaenNnPiQgO0xnu
      ToxutRZJfJvG4Ox4ka3GORQd9CsCZ2vsUDmsXOfUENOyMqADC6p1M3h33tsu
      rY15k9qMSpG9OX_IJAXmxzAh_tWiZOwk2K4yxH9tS3Lq1yX8C1EWmeRDkK2a
      hecG85-oLKQt5VEpWHKmjOi_gJSdSgqcN96X52esAQ",
 "p":"2rnSOV4hKSN8sS4CgcQHFbs08XboFDqKum3sc4h3GRxrTmQdl1ZK9uw-PIHf
      QP0FkxXVrx-WE-ZEbrqivH_2iCLUS7wAl6XvARt1KkIaUxPPSYB9yk31s0Q8
      UK96E3_OrADAYtAJs-M3JxCLfNgqh56HDnETTQhH3rCT5T3yJws",
 "q":"1u_RiFDP7LBYh3N4GXLT9OpSKYP0uQZyiaZwBtOCBNJgQxaj10RWjsZu0c6I
      edis4S7B_coSKB0Kj9PaPaBzg-IySRvvcQuPamQu66riMhjVtG6TlV8CLCYK
      rYl52ziqK0E_ym2QnkwsUX7eYTB7LbAHRK9GqocDE5B0f808I4s",
 "dp":"KkMTWqBUefVwZ2_Dbj1pPQqyHSHjj90L5x_MOzqYAJMcLMZtbUtwKqvVDq3
      tbEo3ZIcohbDtt6SbfmWzggabpQxNxuBpoOOf_a_HgMXK_lhqigI4y_kqS1w
      Y52IwjUn5rgRrJ-yYo1h41KR-vz2pYhEAeYrhttWtxVqLCRViD6c",
 "dq":"AvfS0-gRxvn0bwJoMSnFxYcK1WnuEjQFluMGfwGitQBWtfZ1Er7t1xDkbN9
      GQTB9yqpDoYaN06H7CFtrkxhJIBQaj6nkF5KKS3TQtQ5qCzkOkmxIe3KRbBy
      mXxkb5qwUpX5ELD5xFc6FeiafWYY63TmmEAu_lRFCOJ3xDea-ots",
 "qi":"lSQi-w9CpyUReMErP1RsBLk7wNtOvs5EQpPqmuMvqW57NBUczScEoPwmUqq
      abu9V0-Py4dQ57_bapoKRu1R90bvuFnU63SHWEFglZQvJDMeAvmj4sm-Fp0o
      Yu_neotgQ0hzbI5gry7ajdYy9-2lNx_76aBZoOUu9HCJ-UsfSOI8"
}
```

Figure 4 — JWK example of a KEK (asymmetric key)[5]:

```
{
      "kid": "020e2b52-c793-814a-8526-387ce0571fb4",
      "kty": "RSA",
      "n": "5MPCfUAkhGG6w76Cw2b7vzmyM-K4-80bVn_aPMHHEBa4SQPfERmK_Q4L9fD6FD6krj_
RU_DCYENmMo0ceZQymePdSmeSHgbrkyU9vXfvLDHNftGPgH0xtQmc-gBWKMopRs6Svd13CCFaKn8P
66iF25yVwmc13-5WKGSLJV5oiDa3vOfiJKSqWnZAkejo2BaOSOl9R0qPjLt7z8B18LqTkNeOnsYig
MIeAjis4CrXWVYfbIpryOLFcGBC4gCHiF7tvP5YR3HtqDSmTNzK3xqSFNn_3PMRaGByV8yxcWDB3-
2lRr5JwznuZlm37r_RptgsU73AfhL1phFhYLdTQQ5kmQ",
      "e": "AQAB",
      "use": "enc",
      "key_ops": [
        "wrapKey",
        "unwrapKey"
      ]
}
```

Figure 5 — JWK example of a PK (public part of the asymmetric key):

5.4.2. DEK Specific Properties

For expressing access and use conditions, the standard compliant JWK key representation must be extended with domain specific properties.

The following is JSON compliant encoding of the variables of the class `DEK`. The KMS returns this representation when the `Accept` header has the value `application/json` or `application/jwk+json` or the `f` parameter has the value `JWK`.

[5]Source: RFC 7517, C.1

To obtain a JWKS (JSON Web Key Set) compliant encoding as defined in IETF RFC 7517, the `Accept` header must have the value `application/jwk-set+json` or the `f` parameter must have the value `JWKS`.

```
{
  "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
  "alg": "A128GCM",
  "kty": "oct",
  "k": "J_W99Qhw5gbP72YpmA60Kg",
  "iss": "DCS Service",
  "iat": 1631188397,
  "nbf": 1631189542,
  "naf": 1631210342,
  "active": true,
  "sub": "Long John Silver",
  "aud": [
      "DCS Client"
   ],
  "subs": [
      "Long John Silver",
      "Alice in Wonderland"
  ]
}
```

Figure 6 — Specific properties of the DEK example above

The JWK representation of a DEK contains specific properties as follows.

- `iss`: The actual entity that created the key.
- `iat`: The seconds since the epoch when the key was registered.
- `nbf`: The minimum seconds since the epoch when this key can be accessed via the KMS API.
- `naf`: The maximum seconds since the epoch when this key can be accessed via the KMS API.
- `active`: If true, the key is accessible via the `read()` operation. The use of the `active` enables to quickly disable key access.
- `aud`: The array of application identifiers (e.g. OAuth2 `client_id` values) that are allowed to obtain the key from the KMS.
- `sub`: The owner of the key.
- `subs`: The array of user identifiers (e.g. OAuth2 `user_id` or OpenID Connect `sub` value) that are allowed to obtain the key from the KMS.

5.4.3. JWT Representation of the DEK

The KMS returns a DEK in JWT format if the `Accept` header or the `f` query string parameter has the value `application/jwt`. The JWT representation embeds the JWK representation and adds additional claims to be IETF RFC 7519 compliant. The JWT claims are relevant for controlling access and use of a DEK.

CAUTION

The use of a JWT introduces specific semantics that need to be clarified for the use in the context of KMS.

- *`sub` claim identifies the 'subject' of the JWT. The 'subject' for a JWT that represents a key is the key itself. Therefore, the `sub` value must be the value of the `kid` property.*
- *`iat` claim defines the seconds since the epoch as of when the JWT was created and **not** the key itself. The `iat` property of the key defines the creation (registration) time of the key.*
- *`iss` claim defines the entity that created the JWT and not the key. The `iss` property of the key defines the issuer of the key.*

```
{
  "iss": "KMS",
  "iat": 1631188475,
  "exp": 1631190197,
  "nbf": 1631189542,
  "aud": [
      "DCS Client",
      "019b7173-a9ed-7d9a-70d3-9502ad7c0575"
  ],
  "keys": [
      {
        "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
        "alg": "A128GCM",
        "kty": "oct",
        "k": "J_W99Qhw5gbP72YpmA60Kg",
        "iss": "DCS Service",
        "iat": 1631188397,
        "nbf": 1631189542,
        "naf": 1631210342,
        "active": true,
        "sub": "Long John Silver",
        "aud": [
            "DCS Client"
        ],
        "subs": [
            "Long John Silver",
            "Alice in Wonderland",
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      },
      {
        "kid": "006011ef-1181-492e-bb77-2efb3142c647",
        "alg": "A192CBC-HS384",
        "kty": "oct",
        "k": "lWgm6COZs5mgpDWbhg3gNA",
        "iss": "eb3cacc9-7f06-3af7-8583-ddb68ee1412d",
        "iat": 1637405944,
        "nbf": 1637405944,
        "naf": 1637406243,
        "active": true,
        "sub": "ff1045c2-a6de-31ad-8eb2-2be104fe27ea",
        "aud": [
            "019b7173-a9ed-7d9a-70d3-9502ad7c0575"
        ],
        "subs": [
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      }
  ]
}
```

Figure 7 — Example of JWT including two DEK keys

The semantics of the JWT properties are as follows.

- `iss`: The issuer of the JWT.
- `iat`: The issuing seconds since the epoch when the JWT was issued.
- `exp`: The maximum over all key `expires`.
- `nbf`: The minimum over all key `nbf`.
- `aud`: The union over all key `aud`.

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsImtpZCI6Ijg5M2VmM2M4LWMyNDktNDdhMi05MWUyLTA
wMWEwYjIwMTY0NyIsImprdSI6Imh0dHBzOi8vb2djLnNlY3VyZS1kaW1lbnNpb25zLmNvbS9rbXMva2
VrLzg5M2VmM2M4LWMyNDktNDdhMi05MWUyLTAwMWEwYjIwMTY0NyJ9.eyJpc3MiOiJLTVMiLCJpYXQi
OjE2MzExODg0NzUsImV4cCI6MTYzMTE5MDE5NywibmJmIjoxNjMxMTg5NTQyLCJhdWQiOlsiRENTIEN
saWVudCIsIjAxOWI3MTczLWE5ZWQtN2Q5YS03MGQzLTk1MDJhZDdjMDU3NSJdLCJrZXlzIjpbeyJraW
QiOiIwMDFiZmQzMi0yMmM0LTQ0OTEtOTFlMC0xODg3ZTExZTc0NTMiLCJhbGciOiJBMTI4R0NNIiwia
3R5Ijoib2N0IiwiayI6IkpfVzk5UWh3NWdiUDcyWXBtQTYwS2ciLCJpc3N1ZXIiOiJEQ1MgU2VydmVy
IiwiaXNzdWVkX2F0IjoxNjMxMTg4Mzk3LCJub3RfYmVmb3JlIjoxNjMxMTg5NTQyLCJub3RfYWZ0ZXI
iOjE2MzEyMTAzNDIsImFjdGl2ZSI6dHJ1ZSwic3ViIjoiZmYxMDQ1YzItYTZkZS0zMWFkLThlYjItMm
JlMTA0ZmUyN2VhIiwiYXVkaWVuY2VzIjpbIkRDUyBDbGllbnQiXSwic3ViamVjdHMiOlsiTG9uZyBKb
2huIFNpbHZlciIsIkFsaWNlIGluIFdvbmRlcmxhbmQiLCJmZjEwNDVjMi1hNmRlLTMxYWQtOGViMi0y
YmUxMDRmZTI3ZWEiXX0seyJraWQiOiIwMDYwMTFlZi0xMTgxLTQ5MmUtYmI3Ny0yZWZiMzE0MmM2NDc
iLCJhbGciOiJBMTkyQ0JDLUhTMzg0Iiwia3R5Ijoib2N0IiwiayI6ImxXZ202Q09aczVtZ3BEV2JoZz
NnTkEiLCJpc3N1ZXIiOiJlYjNjYWNjOS03ZjA2LTNhZjctODU4My1kZGI2OGVlMTQxMmQiLCJpc3N1Z
WRfYXQiOjE2Mzc0MDU5NDQsIm5vdF9iZWZvcmUiOjE2Mzc0MDU5NDQsIm5vdF9hZnRlciI6MTYzNzQw
NjI0MywiYWN0aXZlIjp0cnVlLCJzdWIiOiJmZjEwNDVjMi1hNmRlLTMxYWQtOGViMi0yYmUxMDRmZTI
3ZWEiLCJhdWRpZW5jZXMiOlsiMDE5YjcxNzMtYTllZC03ZDlhLTcwZDMtOTUwMmFkN2MwNTc1Il0sIn
N1YmplY3RzIjpbImZmMTA0NWMyLWE2ZGUtMzFhZC04ZWIyLTJiZTEwNGZlMjdlYSJdfV19.Hiyncvcl
Fuw8Lrvx6Yaw2XkvzC75jNt6Le7ekDWjpvlzvxP_ZdQahUbGMe5jidfKQVfKnaPkyqQCeGe1AfMdhJs
Et7BE7uJhFNFvviI_dGLxOwZsfO_wmtpO5GjmP1NQHrAJXiE02dd70087gAcvm88VEgc1SK0kvc4WtW
dujuAVSwdbPGd5Cfy6QvA8ZW80cVGlUScKdnX2rtoC7QvmbiqpeI757GGNhpgBT1drKZp8UoR06Kw8D
pFzXNWdQzkGQ3bVJnue0d9gkofrGtLRE5_Erkokvd0bkfaQfNGCH0J6jYft5sNgYa5jZVeioxim7adL
wUblNZ8T3JBOYSEx0Q
```

Figure 8 — Example: JWT compact serialization of JSON above

5.5. Policy Representation

There is only one official XACML or GeoXACML policy structure defined in XML. To include an XML encoded policy in the JSON representation of a key (`policy` property), one possibility is to use the data URI scheme based on base64 encoding as the value.

```
{
  "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
  "alg": "A128GCM",
  "kty": "oct",
  "k": "J_W99Qhw5gbP72YpmA60Kg",
  "iss": "DCS Service",
  "iat": 1631188397,
  "nbf": 1631189542,
  "naf": 1631210342,
  "active": true,
  "sub": "Long John Silver",
  "aud": [
      "DCS Client"
   ],
  "subs": [
      "Long John Silver",
      "Alice in Wonderland"
  ],
  "policy": "data:application/xacml+xml;base64,PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNv
ZGluZz0iVVRGLTgiPz48IS0tVGhpcyBmaWxlIHdh..."
}
```

Figure 9 — Example DEK representation in JSON including policy

```
namespace KMS {
  import Attributes.*

  attribute key_subs {
    id = "urn:sd:key:subs"
    type = string
    category = resourceCat
  }

  attribute key_aud {
    id = "urn:sd:key:aud"
    type = string
    category = resourceCat
  }

  attribute key_nbf {
    id = "urn:sd:key:not-before"
    type = dateTime
    category = environmentCat
  }

  attribute key_naf {
    id = "urn:sd:key:not-after"
    type = dateTime
    category = environmentCat
  }

  attribute key_active {
    id = "urn:sd:key:active"
    type = boolean
    category = environmentCat
  }

  policy BusinessLogic {

    apply permitOverrides

    rule {
      target
        clause actionId == "read"
          permit
          condition
            key_active == true &&
            stringAtLeastOneMemberOf(subjectId,key_subs) &&
            stringAtLeastOneMemberOf(API.clientId,key_aud) &&
            currentDateTime >= key_nbf &&
            currentDateTime <= key_naf
    }
  }
}
```

Figure 10 — Example of the Integral Business Logic Expressed in ALFA[6]

The policy in XACML3 encoding can be found in Annex B.

[6]See OASIS for more details

6

KEY MANAGEMENT SERVER API

6 KEY MANAGEMENT SERVER API

The Key Management Server consists of an API and a Business Logic that controls CRUD access to the API for managing keys. A key can be seen as a resource (aka `item`), as defined by the OGC API building blocks. In the context of the KMS, an OGC API building block `collection` represents a key of a particular type as defined in Clause 5.1.

6.1. Overview

The KMS described in this document has different functionalities to access keys that are used for different purposes. The following key types are supported by the KMS.

- `DEK` (Data Encryption Key) is used to encrypt and decrypt data.
- `KEK` (Key Encryption Key) is used to encrypt and decrypt a DEK.
- `PK` (Public Key) is used to validate digital signatures on a JWT or to encrypt a DEK.

For each key type a separate functionality is required. This separation leads to different APIs as follows.

- DEK API: The interface to interact with data encryption keys. Endpoints are `/collections/dek/items` and `/collections/dek/items/{kid}`.
- KEK API: The interface to interact with key encryption keys. Endpoints are `/collections/kek/items` and `/collections/kek/items/{kid}`.
- PK API: The interface to interact with public keys. Endpoints are `/collections/pk/items` and `/collections/pk/items/{kid}`.

Interaction diagrams with the KMS interfaces can be found in Annex C.

6.2. Access Token Use

The validation of Bearer or JWT access tokens is limited to the associated audience. According to [rfc6750], an Authorization Server must link an audience with the generated token. Furthermore, a resource server such as the DCS Service and KMS may only accept the token if their `client_id` matches the `aud` from the token. This effectively reduces the re-use of an access token when the DCS Service interacts with the KMS. It is only possible for the DCS

Service to re-use the received token if the KMS audience is also associated with the access token.

As outlined in Clause 8, the Authorization Server should avoid releasing tokens with `aud=['DCS Service','KMS']`. Such audience restrictions requires that the DCS Service exchange the received access token with the Authorization Server to act on behalf of the user. Then, the DCS Service uses its own access token to interact with the KMS. This has the following implications.

- The Authorization Server must never release access tokens where the `aud` contains the KMS and another application.
- The DCS Service, DCS Application, and the KMS must all be registered with the same Authorization Server.
- The DCS Service must use the token-exchange flow as defined in IETF RFC 8693 to obtain an access token.
- The DCS Service needs to add the `DCS Application` explicitly as an audience when registering the generated DEK. To ensure that the DCS Application can obtain the DEK, the DCS Service must set the `sub` explicitly to the `sub` associated with the token received from the DCS Application.

6.3. HTTP Media Types

The KMS supports the following content type values for request and response.

- `application/jwk+json` or `application/json` represents one single key using JWK encoding.
- `application/jwk-set+json` represents a key-set (array of keys) encoded as JWKS.
- `application/jwt` represents a JWK or JWKS as claims of the JWT.
- `application/jose` is the encrypted version of the `application/jwt`.

The KMS also supports the following content type value for the `POST`, `PUT` and `PATCH` requests.

- `application/x-www-form-urlencoded` represents one single key using the `&`-encoding.

A request must either use the HTTP header `Accept` or the GET URL parameter `f` to specify the media type for the return. If both are present (HTTP `Accept` and parameter `f`), the parameter `f` has precedence. The default is `application/jwt`.

6.4. HTTP Methods and KMS Operations

The execution of KMS operations is triggered by a particular HTTP method, but also depends on the presence of certain requested information. The requested information can be submitted differently as the following list indicates.

- as HTTP header, e.g., `Accept`
- as HTTP `GET` parameter encoded as KVP, e.g., `f`
- as HTTP `POST`, `PUT` or `PATCH` message body using Content-Type `application/x-www-form-urlencoded`, e.g., `kid`
- as HTTP `POST`, `PUT` or `PATCH` message body using Content-Type `application/{jwk+json, jwk-set+json}` or `application/jose`, e.g., `{ "kid": "4711"}`
- as part of the HTTP request path, e.g., `/collections/dek/items/{kid}`

Table 5 — Supported HTTP methods for the DEK and KEK endpoints

RESOURCE ENDPOINT	HTTP METHOD				
	GET	POST	PUT	PATCH	DELETE
`/collections/{dek,kek}/items`	bulk read	bulk register or bulk generate	n/a	n/a	n/a
`/collections/{dek,kek}/items/{kid}`	read	n/a	register or generate	update	delete

NOTE All HTTP methods require authentication.

Table 6 — Supported HTTP methods for the PK endpoints

RESOURCE ENDPOINT	HTTP METHOD				
	GET	POST	PUT	PATCH	DELETE
`/collections/pk/items`	bulk read	bulk register	n/a	n/a	n/a
`/collections/pk/items/{kid}`	read	n/a	register	update	delete

NOTE The HTTP method `GET` allows anonymous access.

The logical segregation of the endpoints per key type requires that it is not possible to request `DEK`, `KEK`, and `PK` keys in one request. In particular, the bulk operations are limited to return many keys of the same type. This clear separation is important to ease the definition of the business logic and ensures that a very secure implementation can be achieved as no key type specific logic is mixed. Also, looking at the common use cases, it is never the case that a KEK or PK and a DEK must be requested together. Why? The KEK protects (encrypts) the DEK. So, when the KEK is received, the DEK can be decrypted. Even if the KEK encrypts just the DEK's metadata, an application must first decrypt that DEK metadata to then be able to fetch the DEK based on the decrypted metadata.

6.5. HTTP Status Codes

The KMS uses HTTP status codes in compliance with IETF RFC 7231. The status codes below are of particular importance to the KMS.

Table 7 — General HTTP (error) status codes

HTTP STATUS CODE	HTTP RESPONSE BODY	HTTP HEADER	DESCRIPTION
400	OGC API JSON error	n/a	Request data error
401	n/a	`WWW-Authenticate`	Authentication information missing or invalid
403	n/a	n/a	KMS business logic refuses to process the request
429	n/a	`Retry-After`	KMS instructs the caller to wait x-many seconds
500	JSON	n/a	KMS way to say "I'm sorry"

6.6. KMS API Parameters and Their Meanings

The API parameters represent the data model variables at the HTTP level. In **addition** to the variables defined in the Clause 5.1 section, the following API parameters are also required. These additional parameters mainly carry security context and trigger certain request and response semantics.

Table 8 — Additional API parameters and their definition

PARAMETER	TYPE	DESCRIPTION
`f`	String	The media type of the response
`access_token`	String	Bearer Access Token[a]

[a] The HTTP Authorization header with scheme `Bearer` should be used to avoid token leakage.

See Table 1 for the common parameters.

6.7. KMS API and Operations

The Data Encryption Key (DEK), Key Encryption Key (KEK), and Public Key (PK) APIs are similar in behavior. They support the creating, registration, reading, updating, and deleting of keys for the purpose of encrypting geospatial data. The DEK and KEK API also support key generation.

NOTEThe DEK API operates on **symmetric**, the KEK API operates on **asymmetric**, and the PK API operates on **public** keys. This difference is reflected in the JWK compliant key representation and the associated business logic, not at the API level.

6.7.1. Read

HTTP method `GET` on endpoint `/collections/{dek,kek,pk}/items/{kid}` executes the `read(String kid)` operation. This returns one DEK, KEK, or PK in the requested format.

```
GET /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk+json
Authorization: Bearer 298fj39fh39bf892
```

```
HTTP/1.1 OK
Content-Type: application/jwk+json
Content-Length: 403

{
    "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
    "alg": "A128GCM",
    "kty": "oct",
    "k": "J_W99Qhw5gbP72YpmA60Kg",
    "iss": "DCS Service",
    "iat": 1631188397,
    "nbf": 1631189542,
    "naf": 1631210342,
    "active": true,
    "sub": "Long John Silver",
    "aud": [
        "DCS Application"
    ],
    "subs": [
        "Long John Silver",
        "Alice in Wonderland",
        "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
    ]
}
```

Figure 11 — Example: read of one DEK using `application/jwk+json`

```
GET /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453?public_kid=1234
Host: ogc.secure-dimensions.com/kms
Accept: application/jose
Authorization: Bearer 298fj39fh39bf892


HTTP/1.1 OK
Content-Type: application/jose
Content-Length: 914

eyJhbGciOiJSU0EtT0FFUCIsImVuYyI6IkExMjhHQ00iLCJraWQiOiI0NzExIn0.
IQdC6pONHgOrY_I91q8Ufl8jDYWmF23qX1rsLIbWu5G0SPwdDFZMjWNg1bQlwjyCD_
scGzGhm2MR_UJLikmDiZ6ADvwxOpBUDsiNOFUoGZR-akEU3hMjFLkgxpfNGYOwOh_
PlXj9xUesq6NTUiU875pZgz1cGa5tNGeebme5r8XJnp0Y_bPBV4dvt-
OCXIZ2Sc0JgEQN6I-kw46AeQPHxvzhtuYJVFBJQQ8GCJ01ESxWv0GUx3v_
sEtqqwtINulUf7WbetdIMx8LjslHSNXIMq5yg1KqRMM0iVzzonTBsbDkJwRNA-
9WC2lRmavaF3yh6hYG6X66YoXp7fFhSg_h4Q.dCtk-Abgc3NcKjtAwaKjrQ.y7JM_
UrjGdg0M25sYVdTv2sWoE1HQuEkfp-T1SO3bU_
o9qIRDVYLYHS4Ecr7UuUH7LqPiNO6NCHdHzzWpPGuS3yM_sU-jf-
RwMEOKLpRxMRE3AWBVVcRw1a-RyLEJfnxWIt_BrnAzoz1panyMh_TZ44JQ3ZJSDRgAmYlMj-
sthagph7E7GFQuXKYi0V7MXtvuV_u5h4GWM-Pi-58q3Wk4HgYqByN6AodCN7DlxuRe_
HTsrVA4FQelxZb3c0E2VJ0f76RsYxA0-FWSlWL3UJmC8lsoyaHHjd_
K2xAlzqbQljRztdFDG5KaIP_xSqE5bcnKJHHn64KsLgfza1lVLVmEio7odrIEsPOcdifr_
AwjDdYqj65eQf8WlN1Tpg49sEJBNoihNPJEVH7o03Ivk6MUQ_yFdcVWGkHPlNOnljLHqveSPuxEAmmN
zcxOudl0cYrdVLlTSJxv2gu.t5Iye6YzNKjBkLEc1I5lcg
```

Figure 12 — Example: read of one DEK using `application/jose`

```
{
  "alg": "RSA-OAEP",
  "enc": "A128GCM",
  "kid": "4711"
}
```

Figure 13 — Example JWE header for an encrypted DEK (key wrap)

NOTEThe public key was previously registered with the KMS. The JWE header contains the `kid` to the public key ('4711') which is used to protect the DEK. Decryption of the DEK requires use of the associated private key.

```
GET /collections/kek/items/juliet@capulet.lit
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk-set+json
```

```
Authorization: Bearer 298fj39fh39bf892


HTTP/1.1 200 OK
Content-Type: application/jwk+json
Content-Length: 2179

{
    "keys": [
        {
            "kty":"RSA",
            "kid":"juliet@capulet.lit",
            "use":"enc",
            "n":"t6Q8PWSi1dkJj9hTP8hNYFlvadM7DflW9mWepOJhJ66w7nyoK1gPNqFMSQRy
                O125Gp-TEkodhWr0iujjHVx7BcV0llS4w5ACGgPrcAd6ZcSR0-Iqom-QFcNP
                8Sjg086MwoqQU_LYywlAGZ21WSdS_PERyGFiNnj3QQlO8Yns5jCtLCRwLHL0
                Pb1fEv45AuRIuUfVcPySBWYnDyGxvjYGDSM-AqWS9zIQ2ZilgT-GqUmipg0X
                OC0Cc20rgLe2ymLHjpHciCKVAbY5-L32-lSeZO-Os6U15_aXrk9Gw8cPUaX1
                _I8sLGuSiVdt3C_Fn2PZ3Z8i744FPFGGcG1qs2Wz-Q",
            "e":"AQAB",
            "d":"GRtbIQmhOZtyszfgKdg4u_N-R_mZGU_9k7JQ_jn1DnfTuMdSNprTeaSTyWfS
                NkuaAwnOEbIQVy1IQbWVV25NY3ybc_IhUJtfri7bAXYEReWaCl3hdlPKXy9U
                vqPYGR0kIXTQRqns-dVJ7jahlI7LyckrpTmrM8dWBo4_PMaenNnPiQgO0xnu
                ToxutRZJfJvG4Ox4ka3GORQd9CsCZ2vsUDmsXOfUENOyMqADC6p1M3h33tsu
                rY15k9qMSpG9OX_IJAXmxzAh_tWiZOwk2K4yxH9tS3Lq1yX8C1EWmeRDkK2a
                hecG85-oLKQt5VEpWHKmjOi_gJSdSgqcN96X52esAQ",
            "p":"2rnSOV4hKSN8sS4CgcQHFbs08XboFDqKum3sc4h3GRxrTmQdl1ZK9uw-PIHf
                QP0FkxXVrx-WE-ZEbrqivH_2iCLUS7wAl6XvARt1KkIaUxPPSYB9yk31s0Q8
                UK96E3_OrADAYtAJs-M3JxCLfNgqh56HDnETTQhH3rCT5T3yJws",
            "q":"1u_RiFDP7LBYh3N4GXLT9OpSKYP0uQZyiaZwBtOCBNJgQxaj10RWjsZu0c6I
                edis4S7B_coSKB0Kj9PaPaBzg-IySRvvcQuPamQu66riMhjVtG6TlV8CLCYK
                rYl52ziqK0E_ym2QnkwsUX7eYTB7LbAHRK9GqocDE5B0f808I4s",
            "dp":"KkMTWqBUefVwZ2_Dbj1pPQqyHSHjj90L5x_MOzqYAJMcLMZtbUtwKqvVDq3
                tbEo3ZIcohbDtt6SbfmWzggabpQxNxuBpoOOf_a_HgMXK_lhqigI4y_kqS1w
                Y52IwjUn5rgRrJ-yYo1h41KR-vz2pYhEAeYrhttWtxVqLCRViD6c",
            "dq":"AvfS0-gRxvn0bwJoMSnFxYcK1WnuEjQFluMGfwGitQBWtfZ1Er7t1xDkbN9
                GQTB9yqpDoYaN06H7CFtrkxhJIBQaj6nkF5KKS3TQtQ5qCzkOkmxIe3KRbBy
                mXxkb5qwUpX5ELD5xFc6FeiafWYY63TmmEAu_lRFCOJ3xDea-ots",
            "qi":"lSQi-w9CpyUReMErP1RsBLk7wNtOvs5EQpPqmuMvqW57NBUczScEoPwmUqq
                abu9V0-Py4dQ57_bapoKRu1R90bvuFnU63SHWEFglZQvJDMeAvmj4sm-Fp0o
                Yu_neotgQ0hzbI5gry7ajdYy9-2lNx_76aBZoOUu9HCJ-UsfSOI8"
        }
    ]
}
```

Figure 14 — Example: read of one KEK using **`application/jwk-set+json`**

```
GET /collections/pk/items/020e2b52-c793-814a-8526-387ce0571fb4
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk-set+json
Authorization: Bearer 298fj39fh39bf892


HTTP/1.1 200 OK
Content-Type: application/jwk-set+json
Content-Length: 637

{
    "keys": [
        {
            "kid": "020e2b52-c793-814a-8526-387ce0571fb4",
            "kty": "RSA",
            "n": "5MPCfUAkhGG6w76Cw2b7vzmyM-K4-80bVn_aPMHHEBa4SQPfERmK_
Q4L9fD6FD6krj_RU_DCYENmMo0ceZQymePdSmeSHgbrkyU9vXfvLDHNftGPgH0xtQmc-gBWKMop
Rs6Svd13CCFaKn8P66iF25yVwmc13-5WKGSLJV5oiDa3vOfiJKSqWnZAkejo2BaOSOl9R0qPjLt7
z8B18LqTkNeOnsYigMIeAjis4CrXWVYfbIpryOLFcGBC4gCHiF7tvP5YR3HtqDSmTNzK3xqSFNn_
3PMRaGByV8yxcWDB3-2lRr5JwznuZlm37r_RptgsU73AfhL1phFhYLdTQQ5kmQ",
            "e": "AQAB",
            "use": "enc",
            "key_ops": [
                "wrapKey",
                "unwrapKey"
            ]
        }
    ]
}
```

Figure 15 — Example: read of one PK using `application/jwk-set+json`

For the business logic, please see Clause 5.3.1.

6.7.2. Bulk Read

HTTP method `GET` on endpoint `/collections/{dek,kek,pk}/items` executes the `read(String[] kid)` operation. The `kid` API parameter is a comma-separated list of key identifiers. This returns all keys that the KMS is allowed to return in the requested format.

```
GET /collections/dek/items?kid=001bfd32-22c4-4491-91e0-1887e11e7453,006011ef-
1181-492e-bb77-2efb3142c647 HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk-set+json
Authorization: Bearer 298fj39fh39bf892


HTTP/1.1 200 OK
Content-Type: application/jwk-set+json
Content-Length: 1004

{
   "keys": [
      {
        "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
        "alg": "A128GCM",
        "kty": "oct",
        "k": "J_W99Qhw5gbP72YpmA60Kg",
        "iss": "DCS Service",
        "iat": 1631188397,
        "nbf": 1631189542,
        "naf": 1631210342,
        "active": true,
        "sub": "Long John Silver",
        "aud": [
            "DCS Application"
        ],
        "subs": [
            "Long John Silver",
            "Alice in Wonderland",
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      },
      {
        "kid": "006011ef-1181-492e-bb77-2efb3142c647",
        "alg": "A192CBC-HS384",
        "kty": "oct",
        "k": "lWgm6COZs5mgpDWbhg3gNA",
        "iss": "eb3cacc9-7f06-3af7-8583-ddb68ee1412d",
        "iat": 1637405944,
        "nbf": 1637405944,
        "naf": 1637406243,
        "active": true,
        "sub": "ff1045c2-a6de-31ad-8eb2-2be104fe27ea",
        "aud": [
            "019b7173-a9ed-7d9a-70d3-9502ad7c0575"
        ],
        "subs": [
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      }
    ]
}
```

Figure 16 — Example: read of multiple DEKs

For the business logic, please see Clause 5.3.2.

6.7.3. Register

The register operation supports registering a single DEK with complete representation in one request.

HTTP method `PUT` on endpoint `/collections/{dek,kek,pk}/items/{kid}` executes the `register(String kid)` operation if the HTTP request contains a **full** representation of the key. In particular, the key secret variables as defined in IETF RFC 7518 must be present.

NOTE 1 If the `kid` parameter is not present, but the implementation accepts the request, the implementation must ensure that the execution of the `register(kid)` operation is idempotent! In the case where the `kid` is an auto generated index in the database, the implementation cannot ensure idempotent behavior and must reject the request returning HTTP status code `400`.

Accepted content-types submitted via HTTP header `Content-Type` are as follows.

- `application/x-www-form-urlencoded`: The key parameters are sent form encoded.
- `application/jwk+json`: The key is encoded as JWK.
- `application/jwt`: The JWT payload represents the JSON encoding of the parameters (claims in the JWT). The JWT encoding must only include one key and have the claim `sub` which value is used to register the key.
- `application/jose`: The JWE is the encrypted version of the JWT representation. To ensure that the KMS can decrypt the payload (JWT representation of the key), the KEK expressed in the JWE header using `kid` or `kurl` must be fetchable for the KMS.

NOTE 2The JWT encoding must have the claims `sub` and `iss`. The values are used to register a key when using the Content-Type `application/jwt` or `application/jose`

Table 9 — HTTP status code details for the `register()` operation

HTTP STATUS CODE	HTTP HEADER	DESCRIPTION
201	`Location`	A new key was registered and the `kid` was created by the KMS.
204	n/a	A new key was registered as requested and the URI can be re-used.
303	`Location`	An identical key does already exist but using a different `kid`.
409	n/a	A key with the {`kid`} already exists but has different parameters.

```
PUT /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453 HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk+json
Content-Type: application/jwk+json
Content-Length: 403
Authorization: Bearer 298fj39fh39bf892

{
    "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
    "alg": "A128GCM",
    "kty": "oct",
    "k": "J_W99Qhw5gbP72YpmA60Kg",
    "iss": "DCS Service",
    "iat": 1631188397,
    "nbf": 1631189542,
    "naf": 1631210342,
    "active": true,
    "sub": "Long John Silver",
    "aud": [
        "DCS Application"
    ],
    "subs": [
        "Long John Silver",
        "Alice in Wonderland",
        "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
    ]
}

HTTP/1.1 204 No Content
```

Figure 17 — Example: single DEK registration using content-type `application/jwk+json`

For the business logic, please see Clause 5.3.3.

6.7.4. Bulk Register

The bulk register operation supports registering a set of DEKs with complete representation in one request.

HTTP method POST on endpoint /collections/{dek,kek,pk}/items executes the bulk_register(String []kid) operation. Because a POST request can have any side-effect and is not required to be idempotent, the execution of the bulk_register(String []kid) can register one or many keys. If the kid parameter is not present for a key, an implementation can generate a unique kid without idempotency conditions.

When using the Content-Type application/x-www-form-urlencoded, representing multiple keys is theoretically possible, but such a request structuring is outside the scope of this document. Therefore only sending one key when using the application/x-www-form-urlencoded representation is recommended.

```
POST /collections/dek/items HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk-set+json
Content-Type: application/jwk-set+json
Content-Length: 1004
Authorization: Bearer 298fj39fh39bf892

{
   "keys": [
      {
        "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
        "alg": "A128GCM",
        "kty": "oct",
        "k": "J_W99Qhw5gbP72YpmA60Kg",
        "iss": "DCS Service",
        "iat": 1631188397,
        "nbf": 1631189542,
        "naf": 1631210342,
        "active": true,
        "sub": "Long John Silver",
        "aud": [
            "DCS Application"
        ],
        "subs": [
            "Long John Silver",
            "Alice in Wonderland",
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      },
      {
        "kid": "006011ef-1181-492e-bb77-2efb3142c647",
        "alg": "A192CBC-HS384",
        "kty": "oct",
        "k": "lWgm6COZs5mgpDWbhg3gNA",
        "iss": "eb3cacc9-7f06-3af7-8583-ddb68ee1412d",
        "iat": 1637405944,
        "nbf": 1637405944,
        "naf": 1637406243,
        "active": true,
        "sub": "ff1045c2-a6de-31ad-8eb2-2be104fe27ea",
        "aud": [
            "019b7173-a9ed-7d9a-70d3-9502ad7c0575"
        ],
        "subs": [
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      }
   ]
}

HTTP/1.1 204 No Content
```

Figure 18 — Example: multiple DEK registration using content-type `application/jwk-set+json`

The KMS may return HTTP `204` status on a successful registration. This is because the complete key information was already represented in the request.

The KMS returns HTTP status `400` if the request could not be processed.

```
POST /collections/dek/items HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk-set+json
Content-Type: application/jwk-set+json
Content-Length: 953
Authorization: Bearer 298fj39fh39bf892

{
   "keys": [
      {
        "alg": "A128GCM",
        "kty": "oct",
        "k": "J_W99Qhw5gbP72YpmA60Kg",
        "iss": "DCS Service",
        "iat": 1631188397,
        "nbf": 1631189542,
        "naf": 1631210342,
        "active": true,
        "sub": "Long John Silver",
        "aud": [
            "DCS Application"
        ],
        "subs": [
            "Long John Silver",
            "Alice in Wonderland",
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      },
      {
        "kid": "006011ef-1181-492e-bb77-2efb3142c647",
        "alg": "A192CBC-HS384",
        "kty": "oct",
        "k": "lWgm6COZs5mgpDWbhg3gNA",
        "iss": "eb3cacc9-7f06-3af7-8583-ddb68ee1412d",
        "iat": 1637405944,
        "nbf": 1637405944,
        "naf": 1637406243,
        "active": true,
        "sub": "ff1045c2-a6de-31ad-8eb2-2be104fe27ea",
        "aud": [
            "019b7173-a9ed-7d9a-70d3-9502ad7c0575"
        ],
        "subs": [
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      }
  ]
}
```

```
HTTP/1.1 200 OK
Content-Type: application/jwk-set+json
Content-Length: 163

{
   "keys": [
      {
        "kid": "001bfd32-22c4-4491-91e0-1887e11e7453"
      },
      {
        "kid": "006011ef-1181-492e-bb77-2efb3142c647"
      }
   ]
}
```

Figure 19 — Example: multiple DEK registration using content-type `application/jwk-set+json`

NOTE 1 The `kid` parameter is **missing** for the first key. Therefore, the KMS returns a response body that contains the minimum JWK property for each key.

```
POST /collections/dek/items HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk-set+json
Content-Type: application/x-www-form-urlencoded
Content-Length: 314
Authorization: Bearer 298fj39fh39bf892

    alg=A128GCM&
    kty=oct&
    k=J_W99Qhw5gbP72YpmA60Kg&
    issuer=DCS%20Server&
    issued_at=1631188397&
    not_before=1631189542&
    not_after=1631210342&
    active=true&
    sub=Long%20John%20Silver&
    audiences=DCS%20Client&
    subjects=Long%02John%20Silver&
    subjects=Alice%20in%20Wonderland&
    subjects=ff1045c2-a6de-31ad-8eb2-2be104fe27ea&
```

```
HTTP/1.1 201 Created
Location: /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453
```

Figure 20 — Example: single DEK registration using content-type `application/x-www-form-urlencoded`

NOTE 2 The KMS returns 201 with location URI because the `kid` was generated by the KMS.

For the business logic, please see Clause 5.3.4.

6.7.5. Generate

The KMS executes the `generate()` operation instead of the `register()` operation if the key is not fully represented as defined in IETF RFC 7518. For example, the KMS executes the `generate()` operation and not the `register()` operation for a DEK if the request is missing the parameter `k`.

The `generate()` operation creates a key secret and registers the key. On success, the KMS returns the key's representation according to the `Accept` header or `f` parameter.

> ***IMPORTANT***
>
> *The `generate()` operation is not supported for the PK API. Therefore, the KMS must return HTTP status code `400`.*

```
POST /collections/dek/items HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk+json
Content-Type: application/jwk+json
Content-Length: 222
Authorization: Bearer 298fj39fh39bf892

{
    "alg": "A128GCM",
    "kty": "oct",
    "sub": "Long John Silver",
    "aud": [
        "DCS Application"
    ],
    "subs": [
        "Long John Silver",
        "Alice in Wonderland",
        "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
    ]
}
```

```
HTTP/1.1 200 OK
Content-Type: application/jwk+json
Content-Length: 395

{
    "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
    "alg": "A128GCM",
    "kty": "oct",
    "k": "J_W99Qhw5gbP72YpmA60Kg",
    "iss": "KMS",
    "iat": 1654611110,
    "nbf": 1654611110,
    "naf": 1654614256,
    "active": true,
    "sub": "Long John Silver",
    "aud": [
        "DCS Application"
    ],
    "subs": [
        "Long John Silver",
        "Alice in Wonderland",
        "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
    ]
}
```

Figure 21 — Example: single DEK generation using content-type `application/jwk+json`

For the business logic, please see Clause 5.3.5.

6.7.6. Bulk Generate

The `bulk_generate()` supports the generation of a set of DEKs or KEKs. Identical to the `register()` and `generate()` operations, the `bulk_generate()` is executed if the key is not fully represented.

```
POST /collections/dek/items HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk-set+json
Content-Type: application/jwk-set+json
Content-Length: 515
Authorization: Bearer 298fj39fh39bf892

{
   "keys": [
      {
        "alg": "A128GCM",
        "kty": "oct",
        "iss": "DCS Service",
        "iat": 1631188397,
        "nbf": 1631189542,
        "naf": 1631210342,
        "active": true,
        "sub": "Long John Silver",
        "aud": [
            "DCS Application"
        ],
        "subs": [
            "Long John Silver",
            "Alice in Wonderland",
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      },
      {
        "kid": "006011ef-1181-492e-bb77-2efb3142c647",
        "alg": "A192CBC-HS384"
      }
   ]
}

HTTP/1.1 200 OK
Content-Type: application/jwk-set+json
Content-Length: 1004

{
   "keys": [
      {
        "kid": "001bfd32-22c4-4491-91e0-1887e11e7453",
        "alg": "A128GCM",
        "kty": "oct",
        "k": "J_W99Qhw5gbP72YpmA60Kg",
        "iss": "DCS Service",
        "iat": 1631188397,
        "nbf": 1631189542,
        "naf": 1631210342,
        "active": true,
        "sub": "Long John Silver",
        "aud": [
            "DCS Application"
        ],
        "subs": [
            "Long John Silver",
            "Alice in Wonderland",
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      },
      {
        "kid": "006011ef-1181-492e-bb77-2efb3142c647",
        "alg": "A192CBC-HS384",
        "kty": "oct",
        "k": "lWgm6COZs5mgpDWbhg3gNA",
        "iss": "eb3cacc9-7f06-3af7-8583-ddb68ee1412d",
        "iat": 1637405944,
        "nbf": 1637405944,
        "naf": 1637406243,
        "active": true,
        "sub": "ff1045c2-a6de-31ad-8eb2-2be104fe27ea",
        "aud": [
            "019b7173-a9ed-7d9a-70d3-9502ad7c0575"
        ],
        "subs": [
            "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
        ]
      }
  ]
}
```

Figure 22 — Example: multiple DEK generation using content-type `application/jwk-set+json`

For the business logic, please see Clause 5.3.7.

6.7.7. Update

The KMS executes the `update()` operation to change the access and use conditions for the key represented by `kid`.

> *IMPORTANT*
>
> *The `update()` operation cannot be used to change the key representation.*

HTTP method `PATCH` on endpoint `/collections/{dek,kek,pk}/items/{kid}` executes the `update(String kid)` operation.

Table 10 — Response details for the `update()` operation

HTTP STATUS CODE	HTTP RESPONSE BODY	HTTP HEADER	DESCRIPTION
200	{JSON, JWT, JWE}[a]	n/a	Success and key representation in the response body
204	n/a	`Location` optional	Success
400	OGC API JSON error	n/a	Request to update read-only parameter

NOTE Read-only parameters that cannot be changed: `kid`, `sub`, `iss`, and all key variables defined in IETF RFC 7518.

[a] Type depends on the `Accept` header.

```
PATCH /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453 HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk+json
Content-Type: application/jwk+json
Content-Length: 243
Authorization: Bearer 298fj39fh39bf892

{
    "nbf": 1654611110,
    "naf": 1654614256,
    "active": true,
   "aud": [
        "DCS Application",
        "DCS Service"
    ],
    "subs": [
        "Long John Silver",
        "Alice in Wonderland",
        "ff1045c2-a6de-31ad-8eb2-2be104fe27ea"
    ]
}

HTTP/1.1 204 No Content
```

Figure 23 — Example: successful DEK update using content-type `application/jwk+json`

```
PATCH /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453 HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: application/jwk+json
Content-Type: application/jwk+json
Content-Length: 25
Authorization: Bearer 298fj39fh39bf892

{
    "alg": "A192GCM"
}

HTTP/1.1 400 Bad Request
```

Figure 24 — Example: error DEK update using content-type `application/jwk+json`

For the business logic, please see Clause 5.3.8.

6.7.8. Delete

The KMS executes the `delete()` operation for the given `kid`.

HTTP method `DELETE` on endpoint `/collections/{dek,kek,pk}/items/{kid}` executes the `delete(String kid)` operation.

```
DELETE /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453 HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: *
Authorization: Bearer 298fj39fh39bf892

HTTP/1.1 204 No Content
```

Figure 25 — Example: successful DEK delete

As outlined in the Clause 7, the user associated to the access token is identical to the `key.sub` value. HTTP status code `204` indicates that the request was processed successfully.

```
DELETE /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453 HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: *

HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="KMS"
```

Figure 26 — Example: error DEK delete; not authenticated

As outlined in the Clause 7, authorization information must be presented.

```
DELETE /collections/dek/items/001bfd32-22c4-4491-91e0-1887e11e7453 HTTP/1.1
Host: ogc.secure-dimensions.com/kms
Accept: *
Authorization: Bearer 298fj39fh39bf892

HTTP/1.1 403 Forbidden
```

Figure 27 — Example: error DEK delete; not authorized

As outlined in the Clause 7, the request is rejected with `403` if the user associated with the access token is **not** identical to `key.sub`.

For the business logic, please see Clause 5.3.9.

7

KEY MANAGEMENT SERVER BUSINESS LOGIC

7 KEY MANAGEMENT SERVER BUSINESS LOGIC

The Key Management Server's business logic tightly fits the API and the use cases. The use cases involve the following.

- Reading of keys
- Generation of keys
- Registration of keys
- Updating of keys
- Deleting of keys
- Activating / Deactivating of keys
- Managing the access conditions of keys

One essential part of the business logic is the key data model as defined in section Clause 5.1.

7.1. Integrated Business Logic

The integrated business logic is 'hardcoded' into the implementation. The implemented logic provides the minimum security of the registered keys. The implementation leverages the `DEK`, `KEK`, and `PK` class variables to enforce the following logic.

- `{DEK,KEK}.aud` contains the whitelisting of application identifiers.
- `{DEK,KEK}.subs` contains the whitelisting of user identifiers.
- `{DEK,KEK}.emails` contains the whitelisting of user emails.
- `{DEK,KEK,PK}.nbf` defines the minimum seconds since the epoch.
- `{DEK,KEK,PK}.naf` defines the maximum seconds since epoch.

NOTEThe business logic described in this section is an example and is based on the results from OGC Testbeds 16 (Aleksandar Balaban) and 17 (Aleksandar Balaban, Andreas Matheus).

The `key.issuer` variable can be used to support the authenticity of keys. The issuer of a DEK or KEK typically is an application. The issuer of a PK can either be an application or a user.

7.1.1. Operations on a DEK or KEK

The business logic that defines the access conditions to the DEK and the KEK are identical.

```
IF
  key.aud contains request.client_id &&
  (key.subs contains request.sub || key.emails contains request.user_email) &&
  key.nbf > request.dateTime && key.naf < request.dateTime &&
  key.active
THEN
  read(key.kid)
ELSE
  return error
```

Figure 28 — Read DEK or KEK

```
IF
   request.sub != NULL &&
   request.alg != NULL &&
   IF
        key.type == 'DEK' &&
        request.k != NULL
    THEN
        continue
    ELSE
        return 400
   IF
        key.type == 'KEK' &&
        request.n != NULL &&
        ...
    THEN
        continue
    ELSE
        return 400

THEN
  properties = [
      request.sub,
      request.aud := request.client_id,
      request.nbf := now(), request.naf := NULL,
      request.alg, request.kty, request.enc := 'enc'
      ]
  IF
    request.kid != NULL
  THEN
    kid = request.kid
  ELSE
    kid = random()
  register(kid, properties)
ELSE
  return error
```

Figure 29 — Register DEK or KEK

```
IF
   request.sub != NULL &&
   request.alg != NULL &&
   IF
        key.type == 'DEK' &&
        request.k != NULL
    THEN
        continue
    ELSE
        return 400
   IF
        key.type == 'KEK' &&
        request.n != NULL &&
        ...
    THEN
        continue
    ELSE
        return 400
THEN
  properties = [
      request.sub,
      request.aud := request.client_id,
      request.nbf := now(), request.naf := NULL,
      request.alg, request.kty, request.enc := 'enc'
      ]
  IF
    request.kid != NULL
  THEN
    kid = request.kid
  ELSE
    kid = random()
  generate(kid, properties)
ELSE
  return error
```

Figure 30 — Generate DEK or KEK

```
IF
   request.sub != NULL &&
   request.sub == key.sub &&
THEN
   properties = [
      request.active,
      request.subs,
      request.emails,
      request.aud,
      request.nbf, request.naf
      ]
  IF
    request.kid does exist
  THEN
    update(request.kid, properties)
  ELSE
    return 404
ELSE
  return error
```

Figure 31 — Update DEK or KEK

```
IF
   request.sub != NULL &&
   request.sub == key.sub &&
THEN
   IF
     request.kid does exist
   THEN
     delete(request.kid)
   ELSE
     return 404
ELSE
   return error
```

Figure 32 — Delete DEK or KEK

7.1.2. Operations on a PK

The business logic defining access conditions to the PK operations is more lax compared to the DEK and KEK. This is because the implications of unauthorized access are zero. The main reason is that the KMS only stores the public part of the asynchronous key which can only be used for two things:

- verification of a digital signature (by definition created by the user or application in possession of the associated private key); and
- key encryption by any DCS application (client or server) where the receiving party is in possession of the associated private key.

To support public key rotation using the `nbf` and `naf` time constraints is common practice.

```
IF
  key.nbf > request.dateTime && key.naf < request.dateTime
THEN
  read(key)
ELSE
  return 403
```

Figure 33 — Read PK

```
IF
  request.sub != NULL &&
  request.alg != NULL &&
  request.n != NULL
THEN
  properties = [
      request.nbf := now(), request.naf := NULL,
      request.alg, request.kty, request.enc := 'enc'
      request.n
      ]
  IF
    request.kid != NULL
  THEN
    kid = request.kid
  ELSE
```

```
    kid = random()
  register(kid, properties)
ELSE
  return error
```

Figure 34 — Register PK

```
IF
   request.sub != NULL &&
   request.sub == key.sub &&
THEN
   properties = [
      request.active
      ]
  IF
    request.kid does exist
  THEN
    update(request.kid, properties)
  ELSE
    return 404
ELSE
  return error
```

Figure 35 — Update PK

```
IF
   request.sub != NULL &&
   request.sub == key.sub &&
THEN
   IF
     request.kid does exist
   THEN
     delete(request.kid)
   ELSE
     return 404
ELSE
   return error
```

Figure 36 — Delete PK

7.2. Policy-Based Business Logic

The Policy-based business logic provides the key owner a standards-based solution for specifying access conditions that control the `read(kid)` operation for the `DEK` and `KEK`. The logic could even be extended to cover CRUD for a `kid`.

There is a huge advantage in using the Policy-based access control as the same logic can be used when a `DEK` or `KEK` is requested from the KMS and used in the offline scenario. One example use for the offline case could be a GeoPackage with encrypted data, as defined in the OGC

Disaster Pilot '21 Engineering Report[7] In the GeoPackage case, a JWE representation of the `DEK` is stored in the `gpkg_ext_keys` table. The policy stored with the key determines the access control for the application applying the key to encrypted information stored in the GeoPackage's data table(s).

To ensure interoperability of the policy across different implementations (KMS, client applications, etc.) using a OAISIS XACML 3 or OGC GeoXACML[8] is recommended. An implementation that supports the use of the Policy-based access control must be able to construct (Geo(XACML[9] compliant Authorization Decision Response (ADR) and either be integrated with or have access to the Policy Decision Point functionality that derives the Authorization Decision (AD).

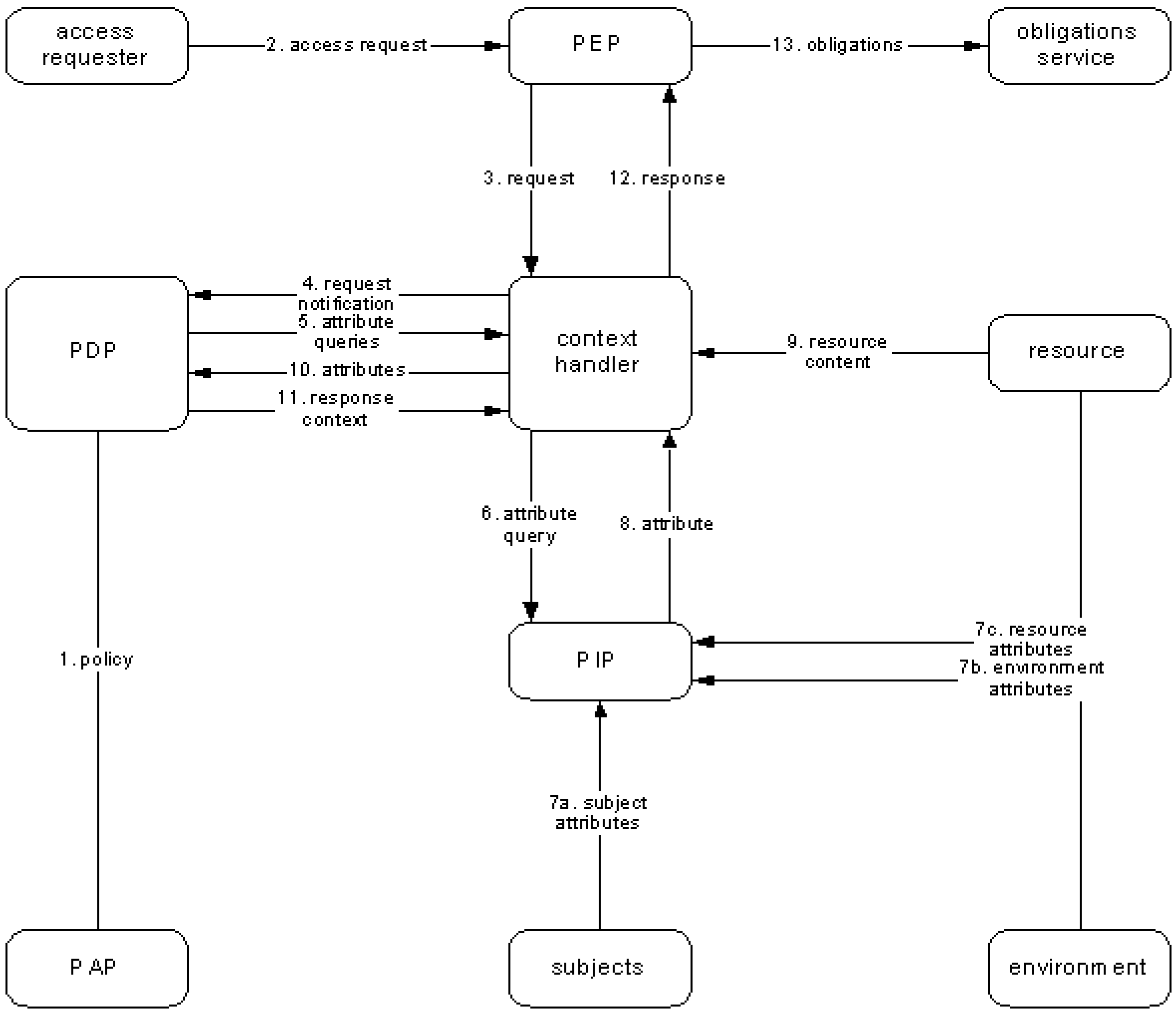


Figure 37 — XACML Flow Diagram

[7]Not yet published at the time of writing.

[8]v3 is currently a work in progress.

[9](Geo)XACML is the abbreviation that means XACML or GeoXACML.

As illustrated in Figure 37, the KMS Policy-based business logic would basically have to implement the following:

- the `PEP` which is the 'gate keeper' that executes the AD; and
- the `context handler` that collects the relevant information to create a XACML standard compliant ADR.

7.3. Key Policy Operations

A KMS business logic that supports the policy-based control for access to a key of type `DEK` or `KEK` must enforce conditions under which it is possible to attach a policy to a key, detach a policy from the key (delete the policy), or replace and update the policy already attached to a key as follows.

- `attachPolicy(kid, policy)`: After a successful execution, the KMS uses the logic from the policy for controlling `read` access to the `DEK` and `KEK`.
- `detachPolicy(kid)`: After a successful execution, the KMS uses the integral business logic for controlling `read` access to the `DEK` and `KEK`.
- `updatePolicy(kid, policy)`: After a successful execution, the KMS immediately uses the renewed logic.

The KMS business logic that controls these operations is an integral part of the implementation. A policy operation is executed if the user is authenticated and can prove ownership of the key as expressed in the following pseudocode.

```
if
  key.sub == request.sub &&
then
  execute policyOperation(...)
else
  deny
```

Figure 38 — Conditions for executing a policy operation

7.4. Implications to the integrated business logic

Once a policy is attached to a key, it overrides the integrated business logic for the `read(kid)` operation on the `DEK` and `KEK`.

8

KMS SECURITY CONSIDERATIONS

8 KMS SECURITY CONSIDERATIONS

Any solution that leverages encryption is only as secure as the entire system, in particular the one managing the encryption keys. Any untrusted implementations that have access to the Key Management Service in particular may jeopardize the entire approach.

This document describes a Key Management System including its API to support a flexible generation, registration, modification, and deletion of encryption keys. The business logic described in section Clause 7 outlines a minimal approach to control access to a key.

One of the paramount requirements for DCS is that the security solution is directly applied to the data without reliance upon the security provided by the underlying communication layer. In that sense, the KMS API supports the exchange of protected (encrypted) keys that are used to encrypt data.

To operate a KMS in a production environment, more requirements must be considered that are beyond the scope of this Testbed 18 activity. Any implementor and operator of a KMS should consider that there are “bad people” out there who will look for the vulnerability in a protocol or a bug in an implementation and, if found, exploit it. The NIST Recommendation for Key Management: Part 1 provides good guidance on how to operate a Key Management Service. The security best practices outlined in T. Lodderstedt et.al. should be considered as well as the recommendations from W3C.

8.1. Encryption is Not a Solution “Until the End of Time”

Any encrypted information can be deciphered and given enough motivation and computing power. Protection by encryption simply gives “you” time to take relevant action such as warning your customers that personal record files have been stolen. But because the credit card information is stored encrypted, your customers have the opportunity to cancel their credit cards and prevent damage before the numbers are deciphered.

8.2. Trusted Applications

For any DCS solution to work securely, it is of paramount importance to also take the client application into the equation. If a client is capable of decrypting information and storing a copy of the original data without protection, the entire solution is questionable. To ensure the release of DEKs and KEKs to client applications, the use of access token audiences should be used. This is one way to establish 3rd party-based trust into client applications. Another model — not considered as the model is only applicable to a certain type of applications — is based on direct trust leveraging digital signatures. Regardless which model is followed, great care must be applied when modifying the `key.audiences` list.

8.3. Vulnerabilities Introduced by Malicious Code

Any Web-browser based application that renders information to the user via HTML is susceptible to malicious JavaScript code injection. If the malicious code intercepts communication or can obtain sensitive information stored in main memory or in browser storage, the leaking of access tokens and, even more dangerous, decryption of data or even DEKs may happen. As soon as Web-browser based clients can be used in "your" federated system, great assurance must exist that **nowhere** in the system malicious code injection is possible! This is a difficult and time-consuming task but paramount when leveraging DCS to protect sensitive data.

8.4. Vulnerabilities Introduced by XSS

Any Web-browser based application must be able to thwart cross-site scripting. Any attacker who can execute XSS is capable of acting as the authenticated user. As such, authorization codes and access tokens can be harvested and replayed to Resource Servers and the Key Management Services without the knowledge of the actual user. As with malicious code injection vulnerability, great care must be taken to ensure that any Web-browser based application is free of XSS vulnerabilities before registration with the Authorization Server.

8.5. Key Delegation

The KMS API supports updating the access conditions for DEK and KEK. Great care must be taken when adding users to the list of trusted `key.subjects` or `key.emails` as well as application identifiers to the `key.audiences`. For example, when delegating key access via emails, only those emails should be used for which the provider guarantees authenticity and uniqueness (another user may never get the same email address some other user had before). User identifiers (subject ids) need to be unique across the entire ecosystem. This is difficult for federated systems. To avoid clashing of user ids, basing uniqueness on DIDs as defined by W3C did-cbor-representation is recommended.

8.6. Use of Policy-Based Access Control

The use of policy-based access control offers great flexibility for the key owner to establish very specific access conditions to a DEK and KEK. Great care must be taken to guarantee that the policy actually permits only "wanted" conditions. A sophisticated test harness should be used to ensure that the policy does not contain “oversights” and produce false positives (unintended permits).

8.7. Deletion of Keys

The NIST Recommendation for Key Management: Part 1 introduces a state model for keys. One of the recommendations is not to prune a key persistently but rather store a key deleted via the API “somewhere else” to make it inaccessible for future access via the API. However, super admin access must be allowed in case an encrypted artifact is discovered that is encrypted with that deleted key.

8.8. Never Exchange Keys in the Clear

The paramount principle of DCS is to ensure integrity and confidentiality to data independent from other security mechanisms. Therefore, the read, generate, and register operations should never exchange DEK or KEK in plain JSON encoding. Following the DCS principle, JWE encoding should always be used.

8.9. Authentication

Any operation on a DEK or KEK requires authentication. If the authentication information is missing or invalid, the implementation should return a HTTP status code `401`.

The PK API’s read operation does not require authentication, as obtaining a public key is not restricted.

8.10. Authorization

The business logic defines the conditions under which a KMS operation is executed for the DEK and KEK. Authorization is always based on authentication which allows identifying the acting user.

To be HTTP compliant, an implementation must return HTTP status code `403` if the execution of the operation is denied.

An implementation may decide to release different details to a user, such as why the execution was rejected, based on the information about the user from the authentication.

8.11. OAuth2 Bearer Token Security Considerations

The use of OAuth2 bearer tokens is widely adopted. However, the ease of use for client applications introduces the greatest danger: A bearer token is like cash — so long it is valid and you know how to use it, you can get basically anything. When an attacker can sniff the network (assuming the Internet threat model), it is possible for them not only to fetch a token but also URLs and HTTP requests that the access token is used with. For OGC APIs with a given path pattern, an attacker could easily use a harvested token and construct valid URLs once one URL was captured. It would thereby be possible in a very short time (perhaps before the token expires) to either fetch all protected resources or, maybe worse, simply delete all resources[10].

As outlined in IETF RFC 6819, a bearer access token can be presented to any Resource Server without additional proof of possession. The use of proof access tokens can thwart the re-use of stolen access tokens. For any proof token, the client must submit some kind of additional knowledge that the access token is being used legitimately. One approach linked to the application is defined in D. Fett et.al.. The proof is via holder-of-key: The use of a private / public key pair is required and the use of an additional HTTP header that contains a digitally signed proof.

It is important that the KMS API avoid key leakage or key management constructed by an attacker. This can be ensured by using DPoP access tokens as defined in D. Fett et.al. on the DCS Service and KMS API. Depending on the HTTP operation and the resource, different options for a DPoP access token can be used. The “simple” proof of processing DPoP HTTP request headers seems to be sufficient when requesting encrypted data from the DCS Service. However, on the KMS API, it is important to thwart request replay. Therefore, the KMS should also leverage the strict resource URI and nonce option. The binding of the resource URI to an access token prevents an attacker from using the access token for …`/items/4711` on …`/items/007`. When fetching a DEK from the KMS, the resource nonce should be used. This ensures that the access token cannot be replayed from the attacker’s system as the nonce is valid only once.

The optional conformance class Annex A.9 allows an implementation to upgrade from Bearer to DPoP access tokens.

[10]Executing a batch script that creates 100s of DELETE requests in parallel is very sufficient!

A

ANNEX A (INFORMATIVE) CONFORMANCE CLASSES

A ANNEX A (INFORMATIVE) CONFORMANCE CLASSES

An implementation of the KMS must implement at least the mandatory conformance class. The implementation of the optional conformance classes would extend the functionality to “deal” with additional use cases.

A.1. Conformance Class DEK (Mandatory)

This conformance class is **mandatory** as it defines the core use of the KMS. An implementation listing the conformance class `DEK` must implement the `DEK` Data Model, Business Logic, and API.

A.2. Conformance Class KEK (Optional)

This conformance class is **optional**. Any implementation listing this conformance class must implement the `KEK` Data Model, Business Logic, and API.

This conformance class supports the creation of JWE encodings for a DEK which is important for use with encrypted GeoPackages.

A.3. Conformance Class PK (Optional)

This conformance class is **optional**. Any implementation listing this conformance class must implement the `PK` Data Model, Business Logic, and API.

This conformance class supports the reading of DEK in JWE format. The client can register the public key to use the key encryption (wrapping).

A.4. OpenAPI (Mandatory)

This conformance class is **mandatory**. Any implementation listing this conformance class must provide an OpenAPI description via the endpoint `/api`.

A.5. Authentication (Mandatory)

This conformance class is **mandatory**. Any implementation listing this conformance class must use the appropriate OpenAPI security scheme to inform clients about the expected authentication protocol.

An implementation must return HTTP status code `401` if required authentication information is missing. The HTTP header `WWW-Authenticate`, as defined in IETF RFC 7235, is used to express further metadata on supported authentication schemes and is recommended.

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm=”KMS", type=“OAuth2”, scope=“DEK” as=“https://
www.authenix.eu”, title=”Access to DEK”
```

Figure A.1 — Example HTTP header WWW-Authenticate for Bearer Access Tokens

The parameters `type`, `scope`, and `as` are KMS specific additions to the basic requirement expressing one authentication scheme (`Bearer` in this case).

A.6. Access Control (Mandatory)

This conformance class is **mandatory**. Any implementation listing this conformance class must implement the integral business logic described in Clause 7.

An implementation must return HTTP status code `403` if the process request is denied by the integral business logic.

A.7. Conformance Class Policy (Optional)

This conformance class is **optional**. Any implementation listing this conformance class must implement the business logic for policy-based access control using XACML 3 or GeoXACML 3 policies.

An implementation must return HTTP status code `403` if the process request is denied by the integral business logic or by the policy-based business logic.

NOTEThe implementation is capable of processing XACML 3 or GeoXACML 3 policies.

A.8. CORS (Optional)

This conformance class is **optional** but recommended. Any implementation listing this conformance class must support W3C cors.

NOTEThe main aspect is that the implementation supports HTTP method `OPTIONS`.

A.9. DPoP (Optional)

This conformance class is **optional** but recommended. As an upgrade from a bearer access token, it prevents the replay of stolen access tokens and thereby prevents the unwanted leakage of DEKs and the management of access conditions for DEKs.

An implementation must be compliant with the Resource Server responsibilities defined in the D. Fett et.al.. It is recommended that the `nonce` option be enforced on the DEK and KEK APIs.

NOTETo prevent the use of access tokens created by an attacker gained via XSS or malicious code injection into a Web-browser based client application (see D. Fett et.al. section 11.4 “Untrusted Code in the Client Context”) specific code review and testing must be applied before such a public client can be registered with the Authorization Server. To prevent an attacker from harvesting authorization codes to independently create access (and refresh) tokens, only Authorization Servers should be used that (i) do not allow the implicit flow and (ii) support DPoP with the authorization request as described in D. Fett et.al. section 10.

B

ANNEX B (INFORMATIVE) POLICY EXAMPLES

ANNEX B (INFORMATIVE) POLICY EXAMPLES

B.1. Example of policy representing the integral business logic

The following example of policy reflects the integral business logic in a XACML based fashion.

```
<xacml3:Policy xmlns:xacml3="urn:oasis:names:tc:xacml:3.0:core:schema:wd-17"
  PolicyId="http://axiomatics.com/alfa/identifier/KMS.BusinessLogic"
  RuleCombiningAlgId="urn:oasis:names:tc:xacml:3.0:rule-combining-algorithm:
permit-overrides"
  Version="1.0">
  <xacml3:Description/>
  <xacml3:PolicyDefaults>
    <xacml3:XPathVersion>http://www.w3.org/TR/1999/REC-xpath-19991116</xacml3:
XPathVersion>
  </xacml3:PolicyDefaults>
  <xacml3:Target/>
  <xacml3:Rule Effect="Permit" RuleId="KMS.BusinessLogic#rule_1">
    <xacml3:Description/>
    <xacml3:Target>
      <xacml3:AnyOf>
        <xacml3:AllOf>
          <xacml3:Match MatchId="urn:oasis:names:tc:xacml:1.0:function:string-
equal">
            <xacml3:AttributeValue DataType="http://www.w3.org/2001/
XMLSchema#string"
              >read</xacml3:AttributeValue>
            <xacml3:AttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:action:action-id"
              Category="urn:oasis:names:tc:xacml:3.0:attribute-category:action"
              DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent=
"false"
            />
          </xacml3:Match>
        </xacml3:AllOf>
      </xacml3:AnyOf>
    </xacml3:Target>
    <xacml3:Condition>
      <xacml3:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
        <xacml3:Apply FunctionId="urn:oasis:names:tc:xacml:3.0:function:any-of-
any">
```

```
          <xacml3:Function
            FunctionId="urn:oasis:names:tc:xacml:1.0:function:boolean-equal"/>
          <xacml3:AttributeDesignator AttributeId="urn:sd:key:active"
            Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
environment"
            DataType="http://www.w3.org/2001/XMLSchema#boolean" MustBePresent=
"false"/>
          <xacml3:AttributeValue DataType="http://www.w3.org/2001/
XMLSchema#boolean"
            >true</xacml3:AttributeValue>
        </xacml3:Apply>
        <xacml3:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
          <xacml3:Apply
            FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-at-least-
one-member-of">
            <xacml3:AttributeDesignator
              AttributeId="urn:oasis:names:tc:xacml:1.0:subject:subject-id"
              Category="urn:oasis:names:tc:xacml:1.0:subject-category:access-
subject"
              DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent=
"false"/>
            <xacml3:AttributeDesignator AttributeId="urn:sd:key:subjects"
              Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
resource"
              DataType="http://www.w3.org/2001/XMLSchema#string" MustBePresent=
"false"
            />
          </xacml3:Apply>
          <xacml3:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:and">
            <xacml3:Apply
              FunctionId="urn:oasis:names:tc:xacml:1.0:function:string-at-
least-one-member-of">
              <xacml3:AttributeDesignator AttributeId="urn:sd:client_id"
                Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
resource"
                DataType="http://www.w3.org/2001/XMLSchema#string"
                MustBePresent="false"/>
              <xacml3:AttributeDesignator AttributeId="urn:sd:key:audiences"
                Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
resource"
                DataType="http://www.w3.org/2001/XMLSchema#string"
                MustBePresent="false"/>
            </xacml3:Apply>
            <xacml3:Apply FunctionId="urn:oasis:names:tc:xacml:1.0:function:
and">
              <xacml3:Apply
                FunctionId="urn:oasis:names:tc:xacml:3.0:function:any-of-any">
                <xacml3:Function
                  FunctionId="urn:oasis:names:tc:xacml:1.0:function:dateTime-
greater-than-or-equal"/>
                <xacml3:AttributeDesignator
                  AttributeId="urn:oasis:names:tc:xacml:1.0:environment:
current-dateTime"
                  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
environment"
                  DataType="http://www.w3.org/2001/XMLSchema#dateTime"
                  MustBePresent="false"/>
                <xacml3:AttributeDesignator AttributeId="urn:sd:key:not-before"
                  Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
environment"
                  DataType="http://www.w3.org/2001/XMLSchema#dateTime"
                  MustBePresent="false"/>
              </xacml3:Apply>
```

```
                <xacml3:Apply
                  FunctionId="urn:oasis:names:tc:xacml:3.0:function:any-of-any">
                  <xacml3:Function
                    FunctionId="urn:oasis:names:tc:xacml:1.0:function:dateTime-
less-than-or-equal"/>
                  <xacml3:AttributeDesignator
                    AttributeId="urn:oasis:names:tc:xacml:1.0:environment:
current-dateTime"
                    Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
environment"
                    DataType="http://www.w3.org/2001/XMLSchema#dateTime"
                    MustBePresent="false"/>
                  <xacml3:AttributeDesignator AttributeId="urn:sd:key:not-after"
                    Category="urn:oasis:names:tc:xacml:3.0:attribute-category:
environment"
                    DataType="http://www.w3.org/2001/XMLSchema#dateTime"
                    MustBePresent="false"/>
                </xacml3:Apply>
              </xacml3:Apply>
            </xacml3:Apply>
          </xacml3:Apply>
        </xacml3:Apply>
      </xacml3:Condition>
    </xacml3:Rule>
</xacml3:Policy>
```

Figure B.1 — XML encoded example XACML3 Policy

```
data:application/xacml+xml;base64,
PD94bWwgdmVyc2lvbj0iMS4wIiBlbmNvZGluZz0iVVRGLTgiPz48IS0tVGhpcyBmaWxlIH
dhcyBnZW5lcmF0ZWQgYnkgdGhlIEFMRkEgUGx1Z2luIGZvciBFY2xpcHNlIGZyb20gQXhp
b21hdGljcyBBQiAoaHR0cDovL3d3dy5heGlvbWF0aWNzLmNvbSkuLS0+PCEtLUFueSBtb2
...
```

Figure B.2 — Base64 encoded policy using data URI scheme (truncated)

NOTEThe use of the data URI scheme maintains the original mime type which is `application/xacml+xml` in this case.

C

ANNEX C (INFORMATIVE) KMS INTERACTION EXAMPLES

C ANNEX C (INFORMATIVE) KMS INTERACTION EXAMPLES

The following diagrams illustrate the basic protocols between DCS components. The purpose for the diagrams is to show the different interactions with the KMS embedded in DCS use cases.

C.1. DCS Consumer Triggers Data Encryption Via DCS Service

The activity starts from the DCS Application where the DCS Consumer requests encrypted data from the DCS Service.

- For the first alternative, the DCS Service generates one or multiple DCSs to process the request. All generated DEKs are registered with the KMS and their reference is included in the DCS response. For a JWE, for example, the header contains the elements `kid` and `kurl`. These elements can be used by the DCS Application to obtain the DEK. Therefore, the DCS Service registers one or many DEKs **on behalf of** the DCS Consumer.

- For the second alternative, the DCS Service receives the DEK reference with the request. The DCS Service uses the DEK fetched from the KMS to encrypt the data. The `kid` and `kurl` elements refer to that DEK.

- For the third alternative, the DCS Application generates a DEK and encrypts it with a KEK. The KEK (a private / public key pair) is registered with the KMS. For a GET request, the DCS Service uses the DEK to encrypt the requested data. For a POST request, the DCS Service uses the unwrapped DEK to decrypt the uploaded data.

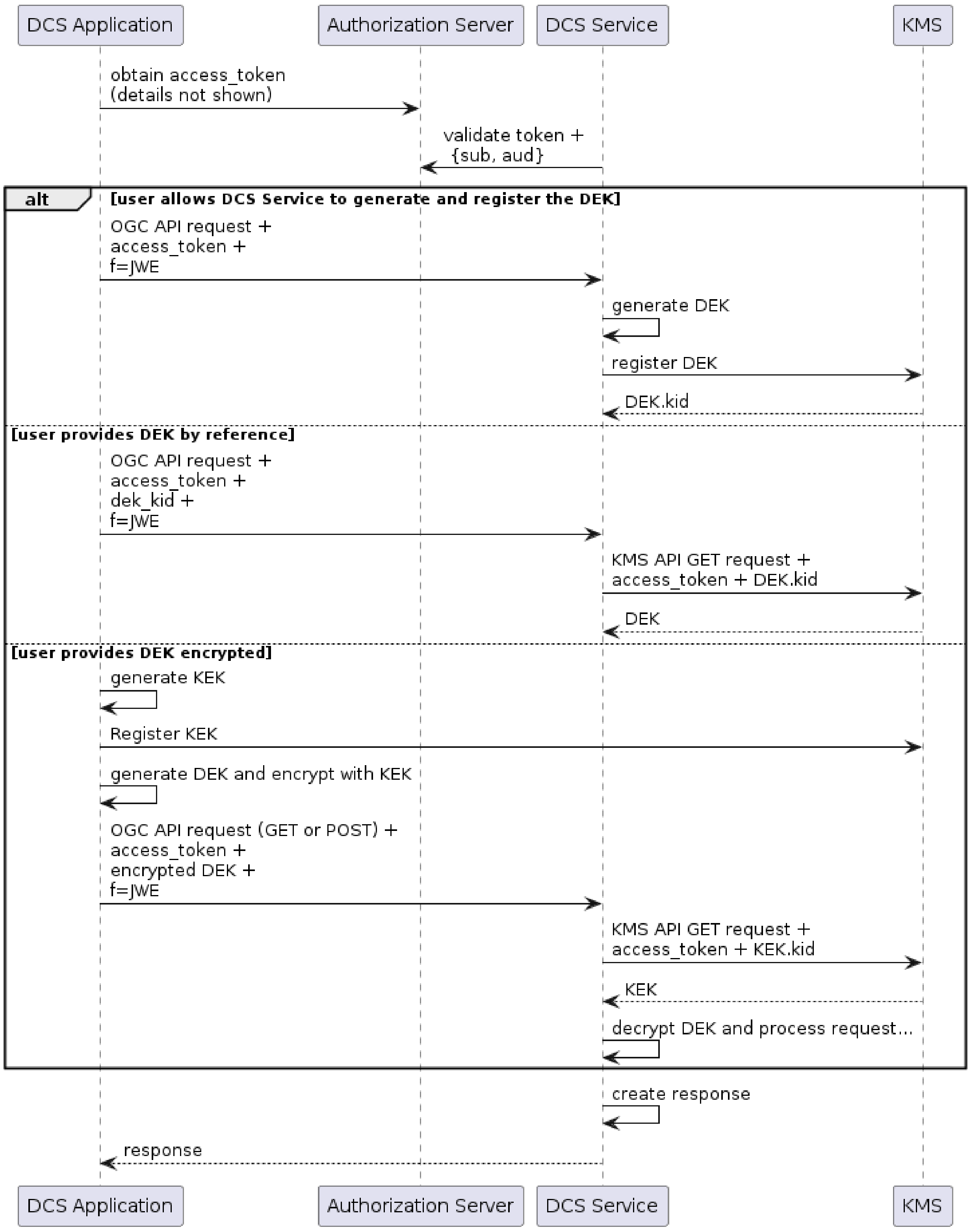

DCS Application
Authorization Server
DCS Service
KMS
obtain access_token (details not shown)
validate token + {sub, aud}
alt
[user allows DCS Service to generate and register the DEK]
OGC API request + access_token + f=JWE
generate DEK
register DEK
DEK.kid
[user provides DEK by reference]
OGC API request + access_token + dek_kid + f=JWE
KMS API GET request + access_token + DEK.kid
DEK
[user provides DEK encrypted]
generate KEK
Register KEK
generate DEK and encrypt with KEK
OGC API request (GET or POST) + access_token + encrypted DEK + f=JWE
KMS API GET request + access_token + KEK.kid
KEK
decrypt DEK and process request...
create response
response

Figure C.1 — Component interactions to request encrypted data (simplified)

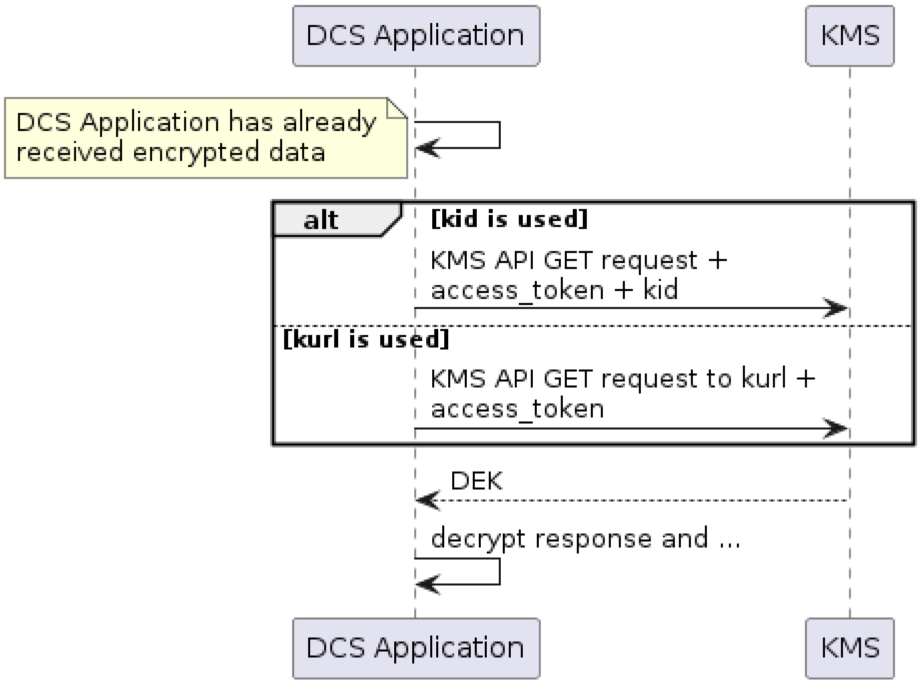

DCS Application
KMS
DCS Application has already received encrypted data
alt
[kid is used]
KMS API GET request + access_token + kid
[kurl is used]
KMS API GET request to kurl + access_token
DEK
decrypt response and ...
DCS Application
KMS

Figure C.2 — Component interactions to decrypt the response (simplified)

C.2. DCS Producer Triggers Data Encryption Via Own DCS Task

The activity starts when a DCS Producer encrypts data in the private network. The generated DEKs get registered with the KMS accessible from an outside network (e.g., the Internet). The DCS Producer uploads the encrypted data to accessible network services, shared drives, etc. or pushes the encrypted data to DCS Consumers. It is also possible to load encrypted data or GeoPackages with encrypted content on mobile devices. At some later time, the DCS Consumer fetches the encrypted data and finds the `kid` or `kurl`. The KMS is contacted to obtain the required DEK. The KMS returns the DEK to the DCS Application if the DCS Producer has set the access conditions accordingly.

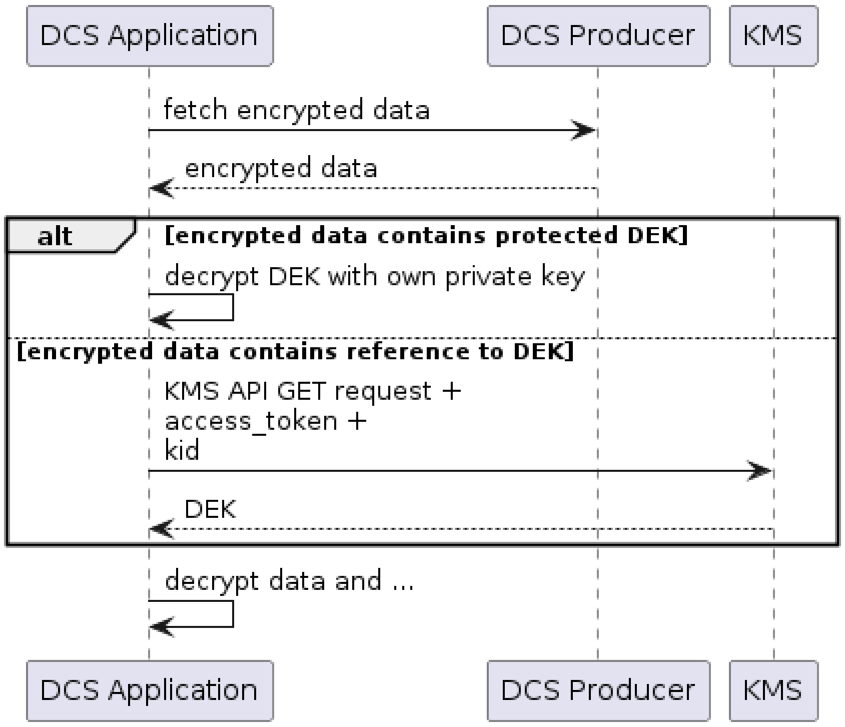

DCS Application
DCS Producer
KMS
fetch encrypted data
encrypted data
alt
[encrypted data contains protected DEK]
decrypt DEK with own private key
[encrypted data contains reference to DEK]
KMS API GET request + access_token + kid
DEK
decrypt data and ...
DCS Application
DCS Producer
KMS

Figure C.3 — Component interactions to decrypt pre-encrypted data (simplified)

C.3. DCS Consumer Triggers Data Signature Via DCS Service

For the first alternative, the DCS Consumer requests data digitally signed by the DCS Service. The DCS Service uses its own private key to produce the digital signature. The DCS Application can fetch the associated public key for signature verification from the `/.well-known/jwks.json `endpoint.

For the second alternative, the DCS Producer can process digitally signed (static) data and make it available to the DCS Consumer(s). In this case, it is assumed that the DCS Producer has no `"my domain"/.well-known` endpoint available. Therefore, the PK gets registered with the KMS. All digitally signed data (e.g., using the JWS encoding) contain the `kid` and `kurl` elements in the header. This enables any application to verify the digital signature by leveraging the `kurl` information. The use of the `kid` assumes an out of band negotiation of the KMS base URL.

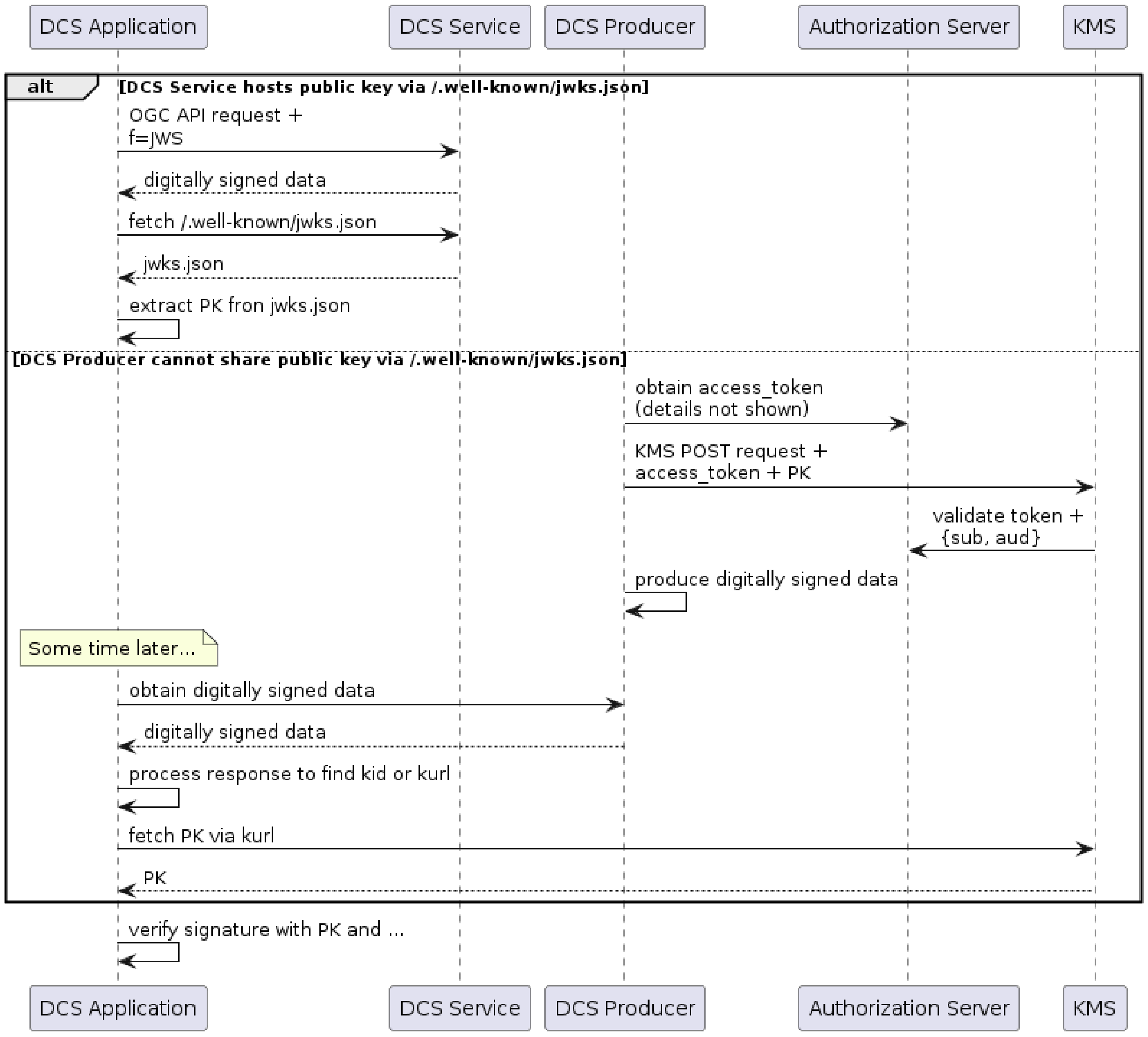

DCS Application
DCS Service
DCS Producer
Authorization Server
KMS
alt
[DCS Service hosts public key via /.well-known/jwks.json]
OGC API request + f=JWS
digitally signed data
fetch /.well-known/jwks.json
jwks.json
extract PK fron jwks.json
[DCS Producer cannot share public key via /.well-known/jwks.json]
obtain access_token (details not shown)
KMS POST request + access_token + PK
validate token + {sub, aud}
produce digitally signed data
Some time later...
obtain digitally signed data
digitally signed data
process response to find kid or kurl
fetch PK via kurl
PK
verify signature with PK and ...

Figure C.4 — Component interactions to verify digitally signed data (simplified)

C.4. Illustration of Access Token Exchange

For the interaction where the DCS Consumer triggers the production of encrypted data, the DCS Service must act on behalf of the user towards the KMS. Assuming access tokens that are properly tied to an aud[11], it is bad practice to make an access token valid for the DCS Service and KMS together. Only a few trusted applications or services should have the privilege of requesting access token from the Authorization Server that are entitled to register (or generate) a DEK with the KMS. In that sense, the DCS Service must exchange the access token received

[11]One or many audiences

from the DCS application into an access token to register DEKs on behalf of the acting user. These interactions are illustrated in the figure below.

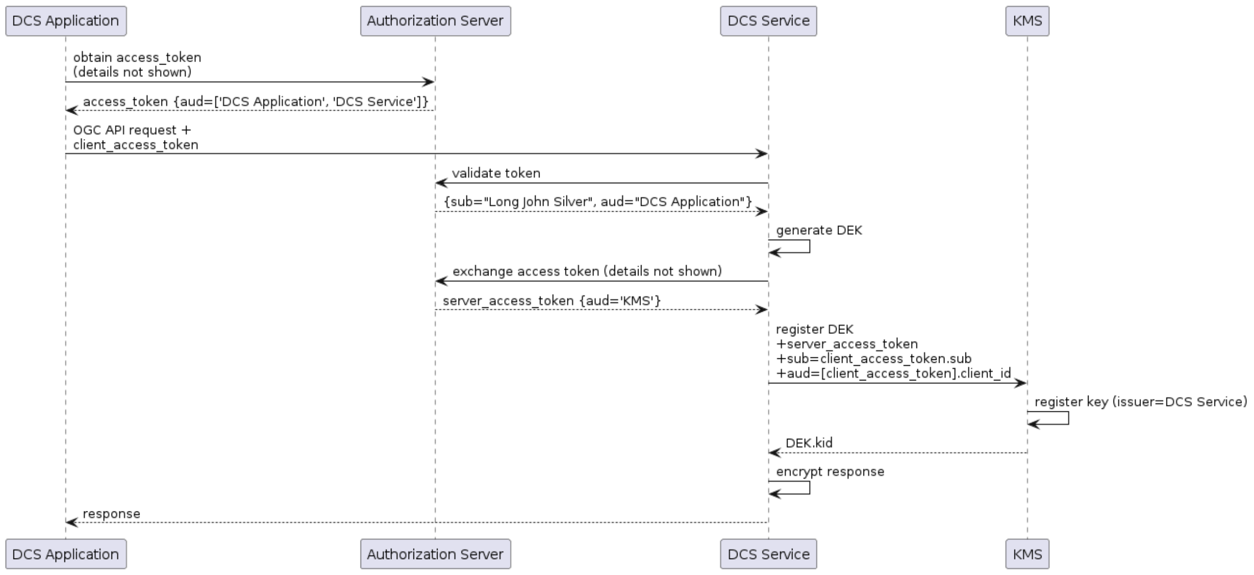

DCS Application
Authorization Server
DCS Service
KMS
obtain access_token (details not shown)
access_token {aud=['DCS Application', 'DCS Service']}
OGC API request + client_access_token
validate token
{sub="Long John Silver", aud="DCS Application"}
generate DEK
exchange access token (details not shown)
server_access_token {aud='KMS'}
register DEK +server_access_token +sub=client_access_token.sub +aud=[client_access_token].client_id
register key (issuer=DCS Service)
DEK.kid
encrypt response
response

Figure C.5 — Example component interactions including token exchange when DCS Service generates DEK and registers DEK with KMS

D

ANNEX D (INFORMATIVE) REVISION HISTORY

D

ANNEX D (INFORMATIVE) REVISION HISTORY

DATE	RELEASE	AUTHOR	PRIMARY CLAUSES MODIFIED	DESCRIPTION
2022-05-31	0.1	A. Matheus	All	Initial version
2022-06-02	0.2	A. Matheus	Mainly section 4 and 5	Drafting data model and DEK API
2022-06-15	0.3	A. Matheus	All main sections	Clarification and soundness
2022-10-19	0.4	A. Matheus	All main sections	Review and completeness, security considerations extended, conformance class DPoP added
2022-10-24	0.5	A. Matheus	All main sections	Integration of Carl Reed comments
2022-11-02	0.6	A. Matheus	All main sections	Integration of Yves Coene comments, example sequence diagrams moved to Appendix C

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Dr. Ron S. Ross, Victoria Y. Pillitteri, Gary Guissanie, Ryan Wagner, Richard Graubart, Deborah Bodeau: NIST SP 800-172, *Enhanced Security Requirements for Protecting Controlled Unclassified Information — A Supplement to NIST Special Publication 800-171*. Gaithersburg, MD (2021). https://csrc.nist.gov/publications/detail/sp/800-172/final.

[2] Abbreviated Language for Authorization Version 1.0, OASIS draft, 2015: https://www.oasis-open.org/committees/download.php/55228/alfa-for-xacml-v1.0-wd01.doc

[3] Web Cryptography API, W3C (2017): https://www.w3.org/TR/WebCryptoAPI/

[4] OGC Testbed-16: Data Centric Security Engineering Report. OGC (2021): https://docs.ogc.org/per/20-021r2.html

[5] OGC Testbed-17: Data Centric Security Engineering Report. OGC (2021): https://docs.ogc.org/per/21-019.html

[6] OAuth 2.0 Demonstrating Proof-of-Possession at the Application Layer (DPoP), IETF draft, 2022 : https://datatracker.ietf.org/doc/html/draft-ietf-oauth-dpop-11

[7] OAuth 2.0 Security Best Current Practice, IETF draft, 2022 : https://www.ietf.org/archive/id/draft-ietf-oauth-security-topics-20.txt

[8] OGC Testbed-18: Secure Asynchronous Catalogs ER. OGC (2022): https://docs.ogc.org/per/22-018.html